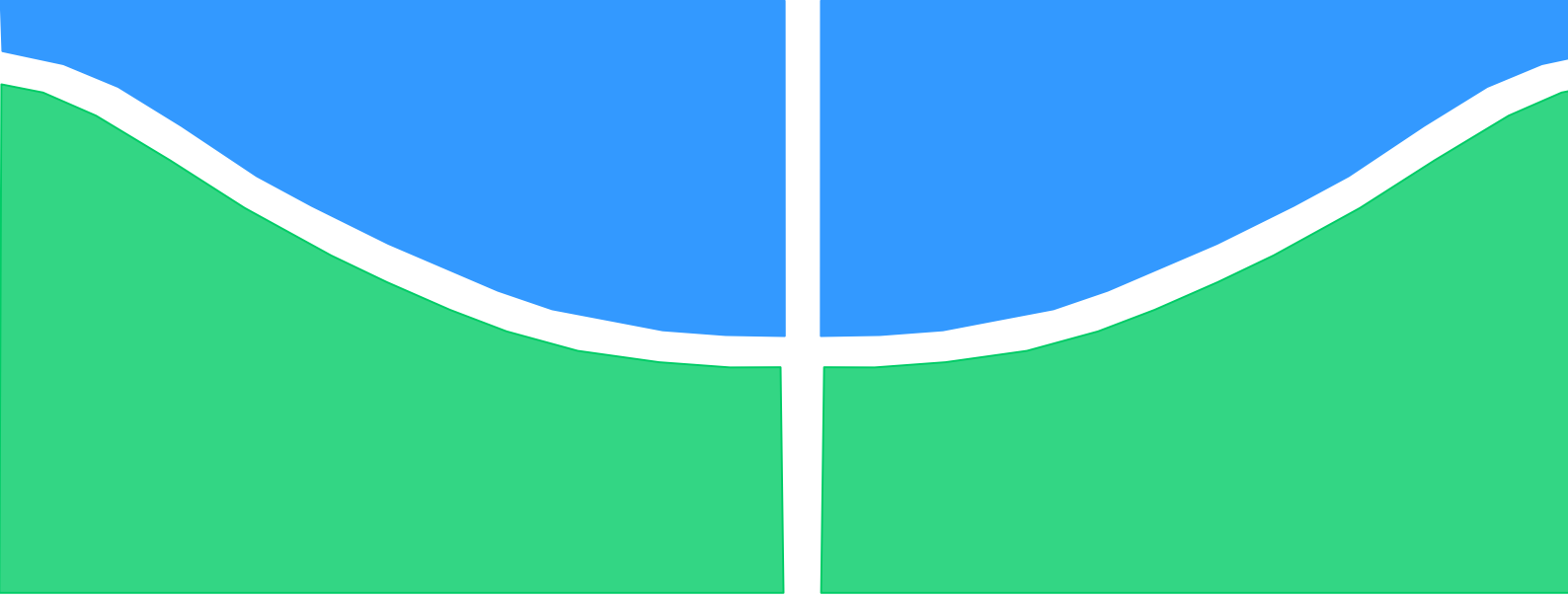


Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

MONITORAMENTO DE QUALIDADE DE SOFTWARE NA ADMINISTRAÇÃO PÚBLICA FEDERAL

Autores: Luiza Maria Pereira Schaidt
Yago Regis Santos Rodrigues
Orientadora: MSc. Elaine Venson

Brasília, DF
2015

LUIZA MARIA PEREIRA SCHAIDT
YAGO REGIS SANTOS RODRIGUES

TÍTULO: MONITORAMENTO DE QUALIDADE DE SOFTWARE NA ADMINISTRAÇÃO PÚBLICA FEDERAL

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Orientadora: MSc. Elaine Venson

Brasília, DF
2015

CIP – Catalogação Internacional da Publicação*

Schaidt, Luiza Maria Pereira e Rodrigues, Yago Regis Santos
Monitoramento de Qualidade de Software na
Administração Pública Federal: Luiza Schaidt, Yago
Rodrigues. Brasília: UnB, 2015. 103 p. : il. ; 29,5 cm.

Monografia (Graduação) – Universidade de Brasília
Faculdade do Gama, Brasília, 2014. Orientação: Elaine Venson.

1. Qualidade de Software. 2. Melhoria de Processos. 3.
Administração Pública I. Venson, Elaine. II.

CDU Classificação

MONITORAMENTO DE QUALIDADE DE SOFTWARE NA ADMINISTRAÇÃO PÚBLICA FEDERAL

Luiza Maria Pereira Schaidt
Yago Regis Santos Rodrigues

Monografia submetida como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software da Faculdade UnB Gama - FGA, da Universidade de Brasília, em 24/06/2015 apresentada e aprovada pela banca examinadora abaixo assinada:

Prof. MSc.: Elaine Venson, UnB/ FGA
Orientador

Prof. Dr.: Luiz Carlos Miyadaira Ribeiro Jr., UnB/ FGA
Membro Convidado

Prof. Dra: Rejane Maria da Costa Figueiredo, UnB/ FGA
Membro Convidado

Brasília, DF
2015

AGRADECIMENTOS

Luiza-

À minha família, em especial a meus pais e a minha irmã, Mônica de Fátima Pereira, Jorge Luiz Schaidt e Clara Pereira Schaidt, pelo apoio e imenso amor a mim dedicados. Esta conquista não seria possível sem seu amparo.

Agradeço a Filipe Barbosa de Almeida, pelo companheirismo e incentivo durante minha graduação.

A Yago Regis Santos Rodrigues, pela dedicação, serenidade e paciência no compartilhamento deste trabalho.

Agradeço imensamente ao corpo docente do curso, em especial a Prof.ª Msc. Elaine Venson, pela generosidade em adotar e orientar este projeto, ao Prof. Dr. Luiz Carlos Miyadaira Ribeiro Jr., pela orientação e entusiasmo que motivou este trabalho.

Aos docentes, Prof.ª Dra. Rejane da Costa Figueiredo, Prof.ª Msc. Cristiane Ramos, Prof.ª Msc. Fabiana Freitas e Prof. Msc. Ricardo Ajax, pelos ensinamentos e por serem verdadeiras inspirações. Meus sinceros agradecimentos e admiração.

Obrigada.

Yago-

Quero agradecer a minha mãe e meu pai, Jurcinéia Santos e Saulo Regis, por me acompanharem por toda a minha vida e por me darem apoio em tudo que decidi fazer até hoje.

Quero agradecer a minha dupla de TCC, por contribuir na evolução do nosso trabalho ao longo desse último ano e não desistir de sempre melhorar o nosso trabalho e a forma com qual trabalhamos.

Agradeço a Prof. ª Msc. Elaine Venson, por toda ajuda oferecida como orientadora e ao Prof. Dr. Luiz Carlos Miyadaira Ribeiro Jr., pela orientação para o começo deste trabalho e por sua continuação em nos ajudar, mesmo não estando na posição de orientador.

Aos docentes, Prof.ª Dra. Rejane da Costa Figueiredo, Prof.ª Msc. Cristiane Ramos, Prof.ª Msc. Fabiana Freitas e Prof. Msc. Ricardo Ajax, pelos ensinamentos e por serem verdadeiras inspirações. Meus sinceros agradecimentos e admiração.

Obrigado.

RESUMO

Qualidade de software, além de influenciar na percepção dos usuários quanto ao produto, pode significar redução de custos em manutenção. Na Administração Pública Federal (APF), onde os serviços de tecnologia da informação são terceirizados, é importante que haja domínio sobre as características de qualidade dos seus sistemas, para auxiliar tomada de decisões e exigir devidamente o nível de qualidade esperado na prestação dos serviços pela empresa contratada. Este trabalho tem por objetivo propor uma solução para monitoramento de qualidade de código na APF, por meio de uma estratégia que considere sobretudo métricas, ferramentas, papéis e procedimentos em um ambiente de integração contínua. Esta estratégia foi elaborada a partir de um estudo de caso em um órgão da APF, e atenta para aspectos importantes dentro de um novo processo de gestão de demanda neste órgão, que adotou a utilização de abordagens ágeis. Este trabalho conta com uma revisão de literatura acerca de temas correlatos tais como Contratações da Administração Pública Federal, Processos de Software, Qualidade de Software, Verificação de Software e Integração Contínua. A definição da estratégia, de forma mais adequada para o contexto do órgão estudado, partiu de um diagnóstico a respeito da qualidade no mesmo. Foram então definidas métricas, ferramentas, papéis e procedimentos de forma a tratar algumas das causas que impactam negativamente a qualidade de software no órgão. A estratégia foi então aplicada e os resultados foram conforme o esperado, ela foi executada com sucesso no órgão. É esperado, para trabalhos futuros a automatização de alguns indicadores que foram sugeridos na estratégia, para então institucionalização da mesma no contexto do órgão.

Palavras-chave: Qualidade de Software. Monitoramento de Qualidade. Administração Pública.

ABSTRACT

Software quality can influence the perception of users about the product an also can mean reduced costs for maintenance. In the Federal Public Administration (APF), where information technology services are outsourced, it is important to have dominion over the quality characteristics of their systems to aid decision-making and properly require the level of quality expected of the services provided by hired company. This study aims to propose a solution for code quality monitoring in the Federal Public Administration (APF), through a strategy that considers mainly metrics, tools, roles and procedures in a continuous integration environment. This strategy has been developed from a case study in an institution of the APF, and attentive to important aspects within a new demand management process this institution, which has adopted the use of Agile approaches. This study includes a literature review about related issues such as Contracts of Federal Public Administration, Software Processes, Software Quality, Software Verification and Continuous Integration. The definition of the strategy, more appropriately for the context of the institution studied, left a diagnosis about the quality in it. They were then defined metrics, tools, roles and procedures in order to treat some of the causes that negatively impact the quality of software in the institution. The strategy was then applied and the results were as expected, it has been successfully performed in the institution. It is hoped, for future work automating some indicators that have been suggested in the strategy, and then institutionalizing it in the context of the institution.

Keywords: *Software Quality. Quality Monitoring. Public Administration.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

.

APF	Administração Pública Federal
BACEN	Banco Central do Brasil
CC	Complexidade Ciclomática
CT	Cobertura de Testes
CMMI	Capability Maturity Model – Integration
DOS	Desconformidade na Ordem de Serviço
GCTI	Gerenciamento de Contrato de Tecnologia da Informação
GeDDAS	Gestão de Demandas de Desenvolvimento Ágil de Software
GQA	Garantia da Qualidade
IAR	Inspeção da Arquitetura de Referência
IDEAL	Initiating, Diagnosing, Establishing, Acting and Learning
IEC	International Engineering Consortium
IEEE	Institute of Electrical and Eletronics Engineers
IN	Instrução Normativa
INEP	Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira
IPE	Índice de Pontos Executados
IPHAN	Instituto do Patrimônio Histórico e Artístico Nacional
ISO	International Organization for Standardization
LOC	Linhas de Código
MCTI	Modelo de Contratação de Soluções de TI
MP	Ministério do Planejamento
MPS.BR	Melhoria de Processo de Software Brasileiro
MVC	Model-View-Controller
NBR	Normas Brasileiras
OS	Ordem de Serviço
PDCA	Plan, Do, Check, Action
PDTI	Plano Diretor de TI
PF	Pontos de Função
PHP	Personal Home Page
SEI	Software Engineering Institute
SEFTI	Secretaria de Fiscalização de Tecnologia da Informação

SISP	Sistema de Administração dos Recursos de Tecnologia da Informação
SLTI	Secretaria de Logística e Tecnologia da Informação
SQuaRE	System and Software Quality Requirements and Evaluation
STF	Supremo Tribunal Federal
TCU	Tribunal de Contas da União
TDD	Test Driven Development
TI	Tecnologia da Informação
TIC	Tencologia da Informação e Comunicações
TST	Tribunal Superior do Trabalho
UnB	Universidade de Brasília
VA	Densidade de Violação de Alta Severidade
VB	Densidade de Violação de Baixa Severidade
VD	Violações de Design
VM	Densidade de Violação de Média Severidade
VX	Densidade de Violção de Muito Alta Severidade
XP	Extreme Programming

SUMÁRIO

1. INTRODUÇÃO

Este primeiro capítulo busca elucidar sobre a visão geral do trabalho, e está dividido em 5 (cinco) partes. A primeira aborda o contexto em que o trabalho está inserido (Contexto), seguido do problema que será abordado (Problema), as justificativas que motivaram o trabalho (Justificativa), os objetivos propostos (Objetivos), os resultados esperados (Resultados Esperados) e, por último, a organização deste trabalho (Organização do Trabalho), que relata a distribuição de conteúdo no decorrer do trabalho em capítulos.

1.1. CONTEXTO

A terceirização de atividades em uma organização permite que a empresa se concentre nas atividades fins de seu produto, isto é, naquilo que ela é capaz de desempenhar com maior qualidade, competitividade e produtividade (PAIVA; SOUZA, 2012). Na Tecnologia da Informação (TI), a adoção da terceirização pode representar benefícios diversos às organizações, proporcionando mais rapidez e baixo custo no acesso a diversas tecnologias. Esses benefícios refletem na melhoria de processos internos e dos serviços prestados ao usuário (PRADO; CRISTOFOLI, 2012).

A Administração Pública Federal (APF) tem determinação prescrita para adquirir serviços de TI por meio de contratações, conforme Decreto-Lei nº 200/6 (BRASIL, 1967), Decreto nº 2.271/97 (BRASIL,1997) e a Instrução Normativa MP/SLTI Nº04 (IN04) (BRASIL, 2014a), uma vez que a TI não é o foco da maioria das instituições federais, e, portanto, não necessita dispor de recursos para desempenhar atividades secundárias à instituição.

Ainda de acordo com a IN04, art. 25, (BRASIL, 2014a), na APF o acompanhamento e garantia da prestação adequada dos serviços e qualidade dos produtos que compõem determinada solução tecnológica é responsabilidade do órgão contratante, e está contida na fase de Gerenciamento de Contrato (BRASIL, 2014). Os processos dessa fase são especificados em um material de referência para consulta, o Guia Prático para Contratações de TI (BRASIL, 2011), onde se encontra o processo de Monitoramento da Execução e suas respectivas atividades, entre elas Avaliar Qualidade, Analisar Desvios de Qualidade.

Atualmente, no entanto, avaliação da qualidade de software na APF, de forma geral, ainda é um grande desafio. A dificuldade em estabelecer diretrizes e processos

adequados dedicados a qualidade é decorrente da falta de domínio dos órgãos em relação ao software em sua completude. Adotar as melhores práticas para gestão das demandas e verificação de software, por exemplo, é uma necessidade latente para melhor aproveitamento do potencial estratégico da TI nas instituições.

1.2. PROBLEMA

Atualmente na APF as instituições que contratam serviços e produtos de TI tem autonomia para contratar também serviços de apoio a qualidade de TI, desde que prestado por uma empresa dedicada exclusivamente a essa função, conforme previsto na legislação (BRASIL, 2014a). Contudo, as formas como as avaliações de qualidade são realizadas em alguns dos órgãos públicos, a exemplo do órgão objeto de estudo deste trabalho (Capítulo 3), são questionáveis por basear-se em *checklists* concentrados em artefatos secundários do software em detrimento do código propriamente dito.

A importância de verificação do código está na proximidade com o produto de software em si. Além disso, Mills (1988) já destacava a importância da objetividade das métricas e sabe-se que a análise de documentos, ainda que definida através de critérios, acaba estando sujeita a análises subjetivas do avaliador. A análise de código permite maior objetividade da avaliação e consequentemente maior confiabilidade das medidas.

A redução do fator humano nas medições permite que medidas sejam automatizadas, sendo menos custosas e geralmente mais rápidas. Nas instituições que desejam utilizar das metodologias ágeis para se adequar ao mercado e atingir objetivos estratégicos, é importante garantir a agilidade também nas medições, para que a análise seja veloz e a homologação do incremento entregue e acompanhamento do projeto como um todo não fiquem defasados em relação ao desenvolvimento do software.

Os benefícios da agilidade nas medições de qualidade se estendem também aos procedimentos de manutenção, que viabilizam a avaliação do código mantido e evita que novos *bugs* ou inconformidades sejam inseridos e consequentemente degradem a qualidade dos softwares da instituição.

O desafio é estabelecer uma Estratégia de Avaliação de Qualidade centrada no código que seja adequada ao contexto de um órgão público federal. Assim, a questão de pesquisa deste trabalho é:

"Como monitorar a Qualidade de Código dos produtos de software adquiridos na Administração Pública Federal? "

1.3. JUSTIFICATIVA

Com a obrigatoriedade de aquisição de serviços denominados comuns por meio de contratações (BRASIL, 1967), tornam-se fundamentais as contratações de serviços de TI na esfera pública. Estabelecer maneiras de aferir a qualidade do produto ou serviço prestado é um desafio atual enfrentado pelos gestores dos órgãos públicos na administração federal.

A terceirização de serviços pode trazer benefícios diversos às organizações, contanto que sejam planejadas e reguladas com rigor. Na APF, a justificativa para terceirizações parte do pressuposto que as soluções terceirizadas sejam passíveis de avaliação objetiva de sua qualidade, para garantir o bom emprego do dinheiro público envolvido. A IN04 (BRASIL, 2014a) prevê a fixação de Critérios de Aceitação na contratação de soluções de TI, que são parâmetros objetivos e mensuráveis utilizados para verificar se um bem ou serviço recebido está em conformidade com os requisitos especificados.

Atualmente grande parte da Esfera Pública carece de melhorias na avaliação de qualidade de software e em seus processos como um todo. Tomando como exemplo o órgão estudado neste trabalho - apresentado no Capitulo 4, Estudo de Caso-, as diretrizes (inclusos critérios objetivos e mensuráveis de aceitação fixados em relação às características internas do software) para serem seguidas quando se trata de processo de avaliação são insatisfatórias, no que diz respeito à aceitação de Software.

Neste contexto, o trabalho se justifica pela elaboração de uma estratégia ágil e objetiva de verificação de qualidade centrada no código, uma vez que a perspectiva ágil de gestão de demandas, cada vez mais adotada nas instituições públicas, necessita de velocidade para avaliação da qualidade dos incrementos de software entregues com maior frequência.

1.4. OBJETIVOS

O objetivo geral deste trabalho é definir e implementar uma estratégia de verificação centrada no código dos softwares adquiridos na APF alinhada a um ciclo de desenvolvimento ágil.

Entre os objetivos específicos estão:

- Realizar um diagnóstico da situação atual da qualidade no órgão estudado;
- Definir a estratégia de verificação;
- Implementar a estratégia proposta em um projeto piloto no órgão estudado.

1.5. RESULTADOS ESPERADOS

Ao final do estudo, espera-se apresentar uma solução adequada para a melhoria do monitoramento da qualidade de software no órgão a ser estudado. Visando avaliações objetivas, que acompanhem as entregas frequentes de incremento dos softwares adquiridos pelo órgão, bem como advindos de manutenções.

Espera-se, desta forma, colaborar para a melhoria dos processos do órgão estudado, particularmente aqueles relacionados a gestão de contrato e qualidade de software. Além disso, espera-se aumentar a transparência sobre as condições dos softwares da instituição, aumentando o domínio do órgão e facilitando a tomada de decisões.

1.6. METODOLOGIA

A metodologia de pesquisa deste trabalho foi classificada quanto à abordagem, natureza, objetivos e procedimentos técnicos, com base na organização proposta por Gerhard e Silveira (2009), e pode ser visualizada mais claramente na figura a seguir (Figura 1).

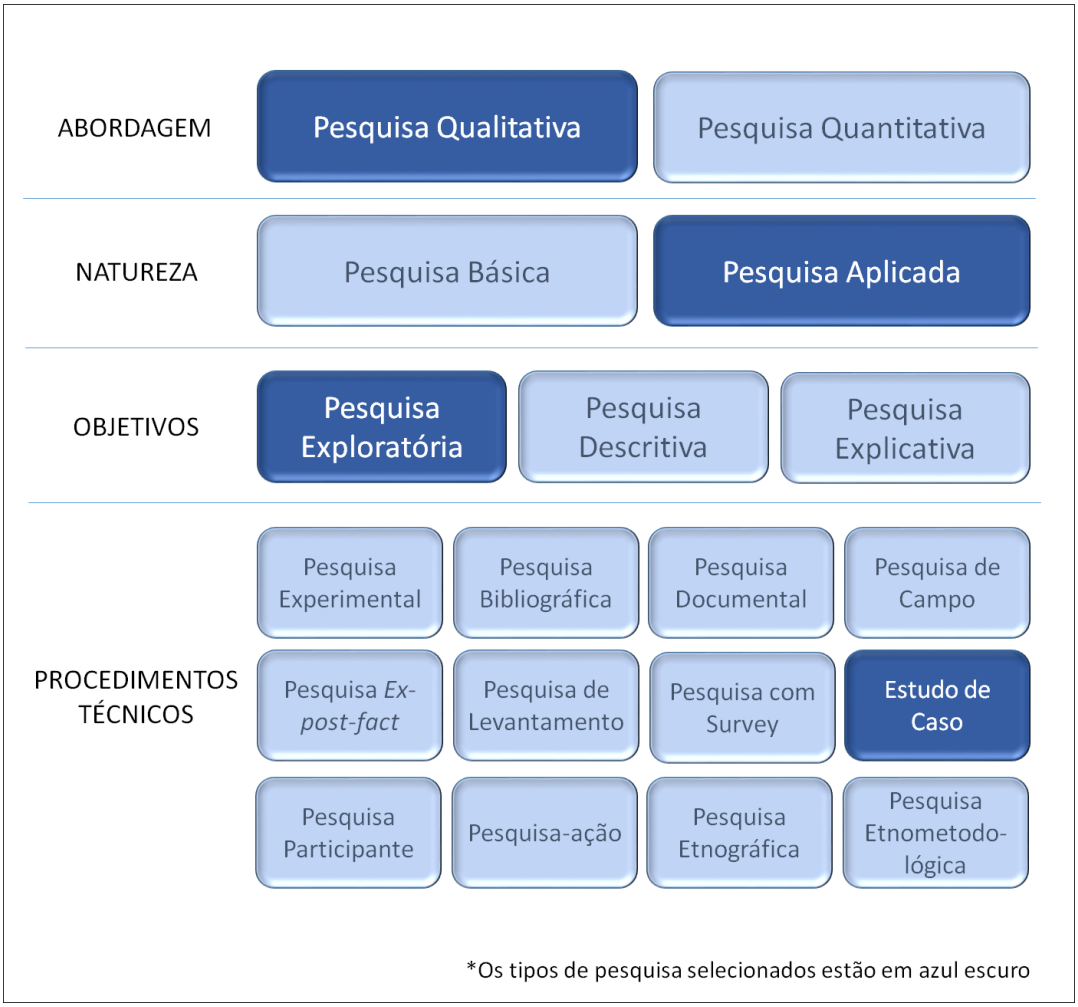


Figura 1- Classificação Metodológica. (Baseado em Gerhard e Silveira, 2009)

A abordagem desta pesquisa é considerada qualitativa, pois apesar de envolver métricas, baseia-se na aceitação da estratégia de verificação de qualidade a ser proposta, ou seja, estão envolvidos valores subjetivos, o que caracteriza a abordagem como qualitativa. No que se refere à natureza, esta pesquisa é aplicada, pois tem como objetivo a elaboração de uma estratégia de verificação da qualidade do produto de software, em um contexto prático, ou seja, conhecimentos dirigidos à solução de problemas específicos (GIL, 2008).

Quanto ao objetivo, o presente trabalho é caracterizado como exploratório, pois tem como foco investigação dos aspectos de implementação que culminem em qualidade de produto nas contratações de soluções de TI. O principal procedimento técnico adotado será um Estudo de Caso, em um órgão da APF.

O plano metodológico adotado será composto por duas fases, a primeira fase, “Iniciação”, de cunho essencialmente acadêmico terá como objetivo principal realizar

um levantamento bibliográfico sobre os temas correlatos à pesquisa. A segunda fase, “Estudo de Caso”, será prática e aplicada ao contexto do Órgão selecionado para o estudo. A figura a seguir, mostra a distribuição das etapas de pesquisa (Figura 2).

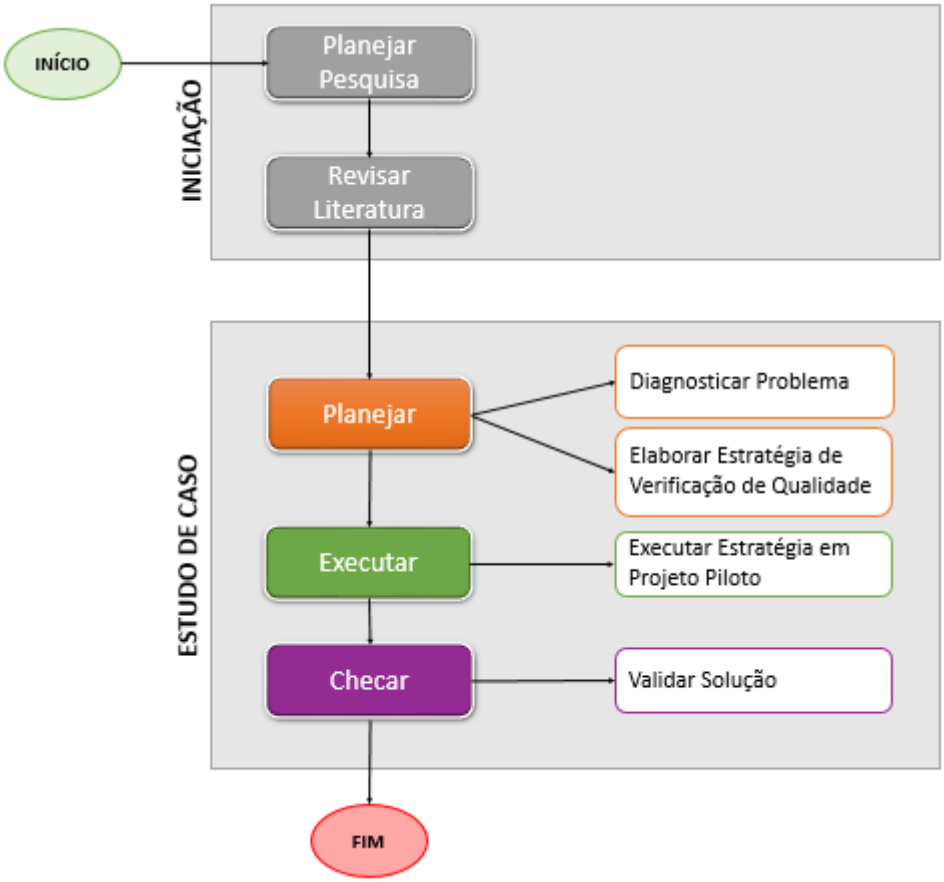


Figura 2- Fases do Estudo.

Na Tabela 1 apresentam-se os procedimentos propostos seguidos dos respectivos objetivos pretendidos de forma mais explícita.

Tabela 1- Etapas e objetivos

Fase 1- Iniciação	
Etapas	**Objetivos**
Planejar Pesquisa	Definição dos objetivos da pesquisa e seleção do tipo de metodologia adotada;
Revisar Literatura	Levantamento bibliográfico com o intuito de estabelecer base teórica para a fundamentação do trabalho

Fase 2- Estudo de Caso	
Etapa	**Objetivos**
Planejar	**Diagnosticar Problema**: Analisar a situação atual do órgão quanto verificação de qualidade de software e identificar prioridades para melhoria. **Elaborar Estratégia de Verificação da Qualidade**: Reunir procedimentos, ferramentas, métricas e papéis que se proponham a atenuar os problemas identificados.
Executar	**Executar Estratégia em Projeto Piloto**: Testar e refinar a solução proposta em um projeto piloto afim de experimentar a solução.
Checar	**Validar Solução**: Analisar a efetividade da solução, com o objetivo de validar a resolução dos problemas identificados.

1.7. ORGANIZAÇÃO DO TRABALHO

Este trabalho divide-se em quatro capítulos: **Introdução**, **Revisão de Literatura**, **Estudo de Caso** e **Considerações Finais**, sendo este o primeiro deles. No capítulo Revisão de Literatura é apresentado um estudo exploratório não sistemático acerca dos temas envolvidos no estudo, são eles Contratações na Administração Pública Federal, Qualidade de Software, Verificação de Software, Integração Contínua e Processo de Software. Em Estudo de Caso é relatada a porção prática do trabalho, aplicado em um órgão da APF. Por fim, no Capítulo Considerações Finais, são apresentadas algumas considerações sobre o trabalho, incluindo sugestões de iniciativas futuras.

2. REVISÃO DE LITERATURA

Este capítulo abrange uma revisão de literatura centrada nos temas: Contratações na Administração Pública Federal, Processo de Software, Qualidade de Software, Verificação de Software e, por fim, Integração Contínua.

2.1. CONTRATAÇÕES NA ADMINISTRAÇÃO PÚBLICA FEDERAL

O Decreto nº 2.271, de 7 de julho de 1997 (BRASIL, 1997) determina que todos os produtos ou serviços que não tiverem ligação direta com a finalidade da instituição do governo federal sejam terceirizados, incluindo entre estes os bens e serviços de tecnologia da informação.

Visando garantir o princípio constitucional da isonomia, as contratações de serviços e produtos no governo brasileiro devem dar prioridade a licitação (BRASIL, 1993). A licitação é o processo formal utilizado pela Administração Pública para publicar, através de edital, sua necessidade de bens e serviços e selecionar a proposta mais vantajosa (BRASIL, 2010).

A licitação possui quatro modalidades: concorrência, tomada de preços, convite e pregão (BRASIL, 2010). O pregão é a forma obrigatória para a contratação de bens e serviços comuns, como definido na Lei nº 10.520 de julho de 2002 (BRASIL, 2002) que o regulamenta.

Em âmbito federal, foi instituído o Sistema de Administração dos Recursos de Tecnologia da Informação (SISP). Seu intuito é organizar a operação, controle, supervisão e coordenação dos recursos de informação e informática dos órgãos públicos federais e está sujeito ao Ministério do Planejamento, Orçamento e Gestão (MP) (BRASIL, 2014c).

O seu principal órgão é a Secretaria de Logística e Tecnologia da Informação (SLTI) do Ministério do Planejamento Orçamento e Gestão (BRASIL, 2014d) que tem como um de seus objetivos normatizar, promover e coordenar ações junto aos órgãos do SISP quanto a gestão e governança de tecnologia da informação; gestão de pessoas e capacitação; e melhoria de processos de desenvolvimento de sistemas. (BRASIL, 2014d)

No uso de suas atribuições a SLTI publicou a Instrução Normativa MP/SLTI Nº04 (IN04) que dispõe sobre o processo de contratação de Soluções de Tecnologia

da Informação pelos órgãos integrantes do Sistema de Administração dos Recursos de Informação e Informática (SISP) do Poder Executivo Federal (BRASIL, 2014a).

A IN04 contempla, entre outras disposições, os procedimentos para a execução das fases de Planejamento da Contratação, Seleção de Fornecedor e Gerenciamento de Contrato (BRASIL, 2014a). Para auxiliar na aderência a legislação brasileira, alguns processos e guias também foram desenvolvidos, como o Guia prático de contratações de TI do Ministério do Planejamento (BRASIL, 2011), Guia de boas práticas em Contratação de Solução de TI do Tribunal de Contas da União (BRASIL, 2012b) e Processo de Contratação de Serviços de TI para Organizações Públicas Federais (CRUZ; ANDRADE; FIGUEIREDO,2011).

2.1.1. Gerenciamento de Contratos

Os procedimentos relacionados a avaliação de qualidade são executados no contexto da fase de Gerenciamento de Contrato (GCTI), que tem por objetivo orientar o domínio do contratante em relação ao serviço sendo prestado, bem como garantir a adequação contratual do mesmo. O art. 25 da IN04 afirma que:

> "...a fase de Gerenciamento do Contrato visa acompanhar e garantir o fornecimento dos bens e a adequada prestação dos serviços que compõem a Solução de Tecnologia da Informação durante todo o período de execução do contrato"
>
> (BRASIL, 2014a. Art. 25).

A fase é composta por cinco processos fundamentais, descritos no Guia prático de contratações de TI do Ministério do Planejamento, são eles (BRASIL, 2011):

- GCTI –P1-Iniciação;
- CGTI –P2-Encaminhar Ordem de Serviço;
- CGTI –P3-Monitoramento da Execução;
- CGTI –P4-Transição Contratual;
- CGTI –P5-Encerramento do Contrato;

A modelagem da fase em questão foi proposta pela SLTI, conforme a Figura 3, e apresenta a relação entre os processos e alguns artefatos esperados.

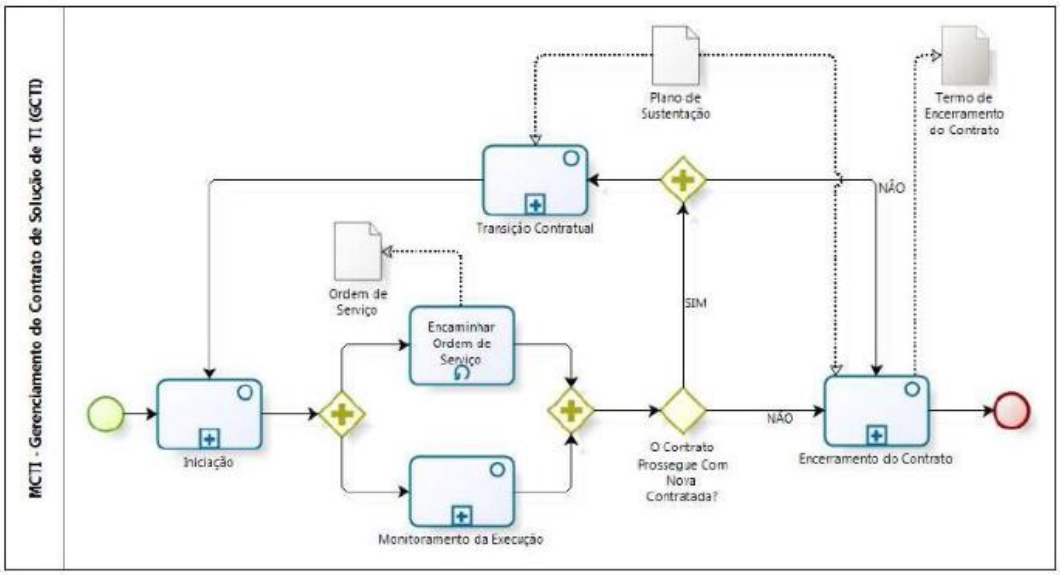


Figura 3. Gestão do Contrato de TI (BRASIL, 2011).

O processo de Monitoramento da Execução (CGTI –P3) é o mais amplo de todo o Modelo de Contratação de Soluções de TI (MCTI), e é composto por quinze atividades. São elas (BRASIL, 2011):

- GCTI-P3.1 – Receber Objeto;
- GCTI-P3.2 – Elaborar Termo de Recebimento Provisório;
- GCTI-P3.3 – Avaliar Qualidade;
- GCTI-P3.4 – Analisar Desvios de Qualidade;
- GCTI-P3.5 – Encaminhar Demandas de Correção;
- GCTI-P3.6 – Efetuar Correções;
- GCTI-P3.7 – Verificar Aderência aos Termos Contratuais;
- GCTI-P3.8 – Indicar Termos Não Aderentes;
- GCTI-P3.9 – Encaminhar Sanções Para Área Administrativa;
- GCTI-P3.10 – Elaborar Termo de Recebimento Definitivo;
- GCTI-P3.11 – Autorizar Emissão de Nota Fiscal;
- GCTI-P3.12 – Emitir Nota Fiscal;
- GCTI-P3.13 – Verificar Irregularidades Fiscais, Trabalhistas e Previdenciárias;
- GCTI-P3.14 – Verificar Manutenção da Necessidade, Economicidade e Oportunidade
- GCTI-P3.15 – Encaminhar Pedido de Alteração Contratual;

Na atividade Avaliar Qualidade (GCTI-P3.3) deverá ser realizada a avaliação da qualidade dos serviços realizados ou dos bens entregues, de acordo com os Critérios de Aceitação previamente definidos.

O Processo de Contratação de Serviços de TI para Organizações Públicas Federais possui uma especificação de tarefas compatíveis com a atividade “Avaliar Qualidade”, que compreende (CRUZ; ANDRADE; FIGUEIREDO,2011):

- GCTI-P3.3.1-Receber os serviços concluídos
- GCTI-P3.3.2-Avaliar os serviços
- GCTI-P3.3.3-Rejeitar serviços inadequados
- GCTI-P3.3.4-Aceitar serviços adequados

Em suma, a partir da análise da legislação e modelos de referência para contratações é possível observar a sistematização já consolidada para os procedimentos relacionados à Avaliação de Qualidade na APF (cerne deste trabalho), suas fases, processos, atividades e tarefas envolvidas.

As fases normatizadas pela IN 04, devem estar presentes nas contratações independente da abordagem que se utilize para gerir as demandas. Atualmente com a crescente na adoção de metodologias ágeis pela APF, surge o desafio de alinhar as determinações legais à velocidade dos processos ágeis.

2.1.2.Metodologias Ágeis na APF

Metodologias ágeis são uma reação aos métodos tradicionais de desenvolvimento de software, eles podem ser compreendidos como um conjunto de práticas que compartilham valores e princípios básicos (COHEN; LINDVALL; COSTA, 2004). A agilidade proposta por estes métodos para Jim Highsmith (2002) significa “a capacidade de criar e responder às mudanças, a fim de lucrar em um ambiente de negócios instável”. A maioria das metodologias ágeis baseia-se na melhoria iterativa, neste sentido uma das metodologias de maior destaque é o *Scrum* (COHEN; LINDVALL; COSTA, 2004), que se trata de um *framework* para a gestão e planejamento de projetos de software que visam entregas rápidas e funcionais em períodos fixos denominados *Sprints* (SCHWABER; BEEDLE 2002).

O setor público tem enfrentado uma forte pressão nas últimas duas décadas para melhorar o seu desempenho no atendimento aos anseios da sociedade

contemporânea (DE BIAZZI; MUSCAT; DE BIAZZI, 2009). Analogamente, desde o início da década de 2000 os métodos ágeis de desenvolvimento de software vêm ganhando crescente popularidade (MELO; FERREIRA, 2010), na esfera pública também estão sendo adaptados com mais frequência especialmente depois do início da década de 2010 (BRASIL, 2013).

Em agosto de 2013, o TCU publicou o Acórdão no 2314/2013 (BRASIL, 2013) que aborda o levantamento elaborado pela Secretaria de Fiscalização de Tecnologia da Informação (SEFTI) acerca do uso de métodos ágeis pelas organizações públicas. O levantamento teve como propósito conhecer as bases teóricas do processo de desenvolvimento de software com métodos ágeis, bem como conhecer experiências práticas de contratação realizadas por instituições públicas federais (BRASIL, 2013). As instituições analisadas nesse levantamento foram: Banco Central do Brasil (BACEN), Instituto do Patrimônio Histórico e Artístico Nacional (IPHAN), Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP), Tribunal Superior do Trabalho (TST) e Supremo Tribunal Federal (STF).

Em linhas gerais, a SEFTI analisou o trabalho desenvolvido nas instituições que utilizam metodologias ágeis de acordo com: métrica para estimativa de tamanho de software utilizada, procedimentos relacionados à gestão das demandas e níveis de serviços estabelecidos. Concentraremos o trabalho na análise de níveis de serviços, que incorporam a avaliação de qualidade nos órgãos. Os pormenores sobre a aferição de qualidade nos órgãos não foram divulgados para manutenção de sigilo estratégico das instituições.

Quanto à aferição da qualidade, o TST criou o indicador Desconformidade na Ordem de Serviço (DOS) para assegurar a conformidade das Ordens de serviço aos requisitos de qualidade, determinados nos modelos a serem utilizados para a elaboração dos artefatos produzidos pela contratada, bem como nas listas de verificação para a avaliação desses artefatos. A meta do indicador DOS é 0%, significando que caso alguma desconformidade seja detectada, o produto, e, consequentemente, a Ordem de serviço, é rejeitada (BRASIL, 2013).

No BACEN para aferir a qualidade dos artefatos entregues pela contratada em projetos de construção, foi definido o Índice de Defeitos por Pontos de Função da OS e o Índice de Defeitos Impeditivos por Pontos de Função da OS. Para o primeiro indicador, há tolerância para a aceitação da OS, enquanto para o segundo a tolerância é zero (BRASIL, 2013).

O Iphan institui apenas dois indicadores de nível de serviço, sendo que somente um deles refere-se ao serviço de desenvolvimento, executado nos ciclos de desenvolvimento, o qual é denominado de Índice de Pontos Executados (IPE). O IPE destina-se a avaliar a qualidade dos produtos entregues, medindo a quantidade de pontos de função brutos executados e aceitos da ordem de serviço. Caso o IPE seja inferior ou igual a 80%, multas podem ser aplicadas à contratada (BRASIL, 2013).

No Inep, os critérios de avaliação da qualidade dos produtos entregues possuem indicadores distintos para a avaliação a cargo da área de TI e a da área de negócios. A área de TI avalia a conformidade da qualidade dos produtos sob o aspecto técnico, homologando-os tecnicamente. A área de negócios avalia a conformidade dos produtos sob o aspecto negocial, homologando-os negocialmente (BRASIL, 2013).

Por fim, o STF aplica dois tipos de redutores às Ordens de serviço pagas à empresa contratada. O primeiro refere-se aos níveis de serviço propriamente ditos, enquanto o segundo relaciona-se aos critérios gerais de avaliação. Entre os níveis de serviço, encontra-se o Índice de Desconformidade do Produto, o qual afere a qualidade dos produtos entregues pela contratada. Por sua vez, nos critérios gerais de avaliação consta a punição a ser aplicada à contratada quando ocorrer atraso não justificado nas datas definidas nas Ordens de Serviço (BRASIL, 2013).

O TCU mapeou os riscos relacionados à adoção de metodologias ágeis no governo federal descritas no levantamento da SEFTI e concluiu, por meio do acórdão, que apesar das divergências entre contratações de TI pela APF e metodologias ágeis, é possível adequar a utilização de metodologias ágeis com as obrigações legais que regulamentam os órgãos públicos federais (BRASIL, 2013).

2.2. PROCESSO DE SOFTWARE

Segundo o MPS.BR(MPS.BR-SW, 2012), processo é um conjunto de atividades inter-relacionadas ou interativas, que transformam insumos (entradas) em produtos (saídas).

Ainda sobre definição de processo, é “um conjunto coerente de procedimentos, tecnologias, artefatos e estruturas organizacionais necessárias a conceber, desenvolver, implantar e manter um produto de software” (FUGGETTA, 2000).

Um processo é o conjunto total de atividades de engenharia necessárias para transformar requisitos do usuário em Software (HUMPHREY, 1989). Um processo determina práticas a serem realizadas pela equipe com prazos definidos e métricas para se avaliar como elas estão sendo realizadas. Define quem, como e quando fazer.

2.2.1.Importância de um Processo de Software

A qualidade de um processo de um sistema de Software é governada pela qualidade do processo usado para desenvolvê-lo (HUMPHREY, 1989).

Durante o desenvolvimento, não é apenas a codificação e as ferramentas que vão ter efeito sobre o produto final (FUGGETTA, 2000), mas também o esforço coletivo e criativo para se obter o produto final. Isto inclui todas as fases do desenvolvimento conhecidas atualmente, desde o levantamento de requisitos até a implantação da solução de Software. Durante a história do Software, alguns problemas fizeram com que a melhoria de processo se tornasse destaque, são eles (FUGGETTA, 2000):

- Produtos entregues que não exibem a qualidade desejada em termos de confiabilidade, funcionalidade ou performance;
- Retardo e trabalho desnecessários em função de uma sequência específica de operações de processos inadequada, que podem ser eliminados ou pelo menos reduzidos pela redistribuição de responsabilidades e atribuições de tarefas;
- Dificuldade de localizar e seguir mudanças e variações de produtos de software gerados por diferentes membros do time de desenvolvimento;
- Grande parte do esforço empregado na produção de software pode ser consumido no próprio esforço para fazer os métodos de desenvolvimento funcionarem (BUTTON; SHARROCK, 1994).

A partir daí a melhoria de processo começou a ganhar grande espaço entre os Engenheiros de Software, e a definição de processo é o primeiro destes passos.

Uma organização possuir um processo de Software definido e documentado tem algumas vantagens, por exemplo, atestar a previsibilidade na hora de produzir produtos de Software, tendo garantia de que seus produtos saíram com qualidade parecida e até igual.

Outra vantagem é, caso tenha certificados, como em ISO ou níveis de maturidade atribuídos, como no *Capability Maturity Model Integration* (CMMI), ter vantagens para aquisição de contratos, já que estas qualificações trazem mais segurança na hora de contratar a empresa, pois já foi provado que ela tem um processo capaz, em algum nível.

Há também, com um processo definido, a capacidade de avaliação dele para implementação de melhorias contínuas, seja em tempo de execução de um projeto, ou com um especialista, interno ou externo à organização, analisando se as melhores práticas de mercado estão presentes ou não no processo.

2.2.2. Componentes de um Processo de Software

Um processo de Software precisa, necessariamente, de algumas coisas para ser considerado completo (CHEMUTURI, 2010), são elas:

- **Procedimentos**: Os procedimentos que são feitos na hora de definir um processo, são o passo a passo para cada tarefa que a organização precisa realizar;
- **Padrões ou Guias**: São aqueles documentos com informações que se aplicam à toda empresa ou organização, relacionada à alguma atividade, por exemplo: padrão de implementação de código Java, Guia para execução de Test Driven Development (TDD) em Java.
- **Templates**: São os padrões de documentos definidos para a empresa/organização, por exemplo: Template para Documento de Visão, Template para Documento de Requisitos.
- **Checklists**: São questionários, com questões de resposta sim, não ou Não se Aplica, que os profissionais utilizam para conferir se a atividade que estão fazendo foi totalmente completada. É utilizado muito quando o processo ainda é recente.

Neste trabalho há mais preocupação com a melhoria de processo, visto que no projeto piloto o processo já está definido, mas há pontos nele e na própria organização que precisam ser melhorados.

2.2.3.Melhoria de Processo de Software

Melhoria de processo é uma ação executada para mudar os processos de uma organização para que eles sigam as necessidades de negócio da organização, permitindo que ela alcance suas metas de negócio mais efetivamente (SECCOM, 2009).

A melhoria de processo de Software para o SISP (SISP, 2011) representa o conjunto de ações para a melhoria dos processos dos órgãos com o objetivo de aperfeiçoar o planejamento, execução e o acompanhamento/monitoramento das atividades da área de TI, com foco na qualidade dos serviços de software.

Há alguns métodos pelos quais pode-se aplicar melhoria de processos em um projeto. Dois deles são o PDCA (Plan, Do, Check, Action) e o IDEAL (Initiating, Diagnosing, Establishing, Acting and Learning). Ambos serão estudados mais a fundo neste trabalho.

2.2.3.1 PDCA

PDCA é um acrônimo para Plan, Do, Check, Action, também é chamado de Ciclo de Deming que é um modelo de gestão de qualidade clássico promovido e praticado no Japão pelo Dr. W. Edwards Deming. A ISO 9001:2000 afirma que: “O método PDCA está disponível para utilização em todos os processos. Enquanto todos os produtos são resultados do processo, a qualidade do produto está relacionada com seu processo de criação” (DERN, 1982). A figura a seguir mostra o ciclo de PDCA, e claramente que o processo de melhoria da qualidade é um processo de ciclo contínuo, na Figura 4. Além disto, este ciclo tem sido utilizado constantemente em empresas para gestão de qualidade e tem se tornado um processo de trabalho lógico que pode ser aplicado em qualquer atividade (WANG, LIU, 2005).

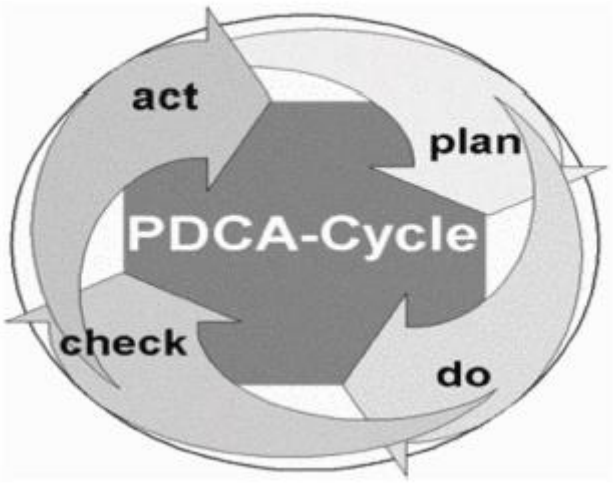


Figura 4. Ciclo de Vida PDCA (WANG, LIU, 2005).

Abaixo uma descrição de cada atividade do PDCA (WANG, LIU, 2005):

- **Plan** - Aqui está a parte de planejamento, os problemas encontrados e as causas. A partir daí os objetivos são definidos e o plano de ação é feito.
- **Do** - Aqui os planos são executados. Caso necessário, algum treinamento é dado antes da execução.
- **Check** - Aqui ocorre a avaliação do que foi feito no Do, os resultados são coletados e verificados.
- **ActIon** - Aqui a ação apropriada é tomada de acordo com os resultados coletados no Check, geralmente se há sucesso a solução é padronizada e tenta disseminação para o resto da empresa, caso haja insucesso ações corretivas são tomadas e parte-se para o próximo ciclo do PDCA.

2.2.3.2 IDEAL

IDEAL é um modelo de melhoria de processo de software (SPI), publicado em 1996 pelo Instituto de Engenharia de Software (SEI) (MCFEELEY, 1996). O objetivo do SPI é melhorar o desenvolvimento de Software O Guia IDEAL são para aquelas empresas que querem iniciar no SPI, ou até mesmo aquelas que já possuem um SPI contínuo (MCFEELEY, 1996).

O modelo IDEAL é composto por 5 fases com descrições genéricas e uma descrição de passos recomendados. A Figura 5, a seguir, mostra a composição do ciclo do IDEAL.

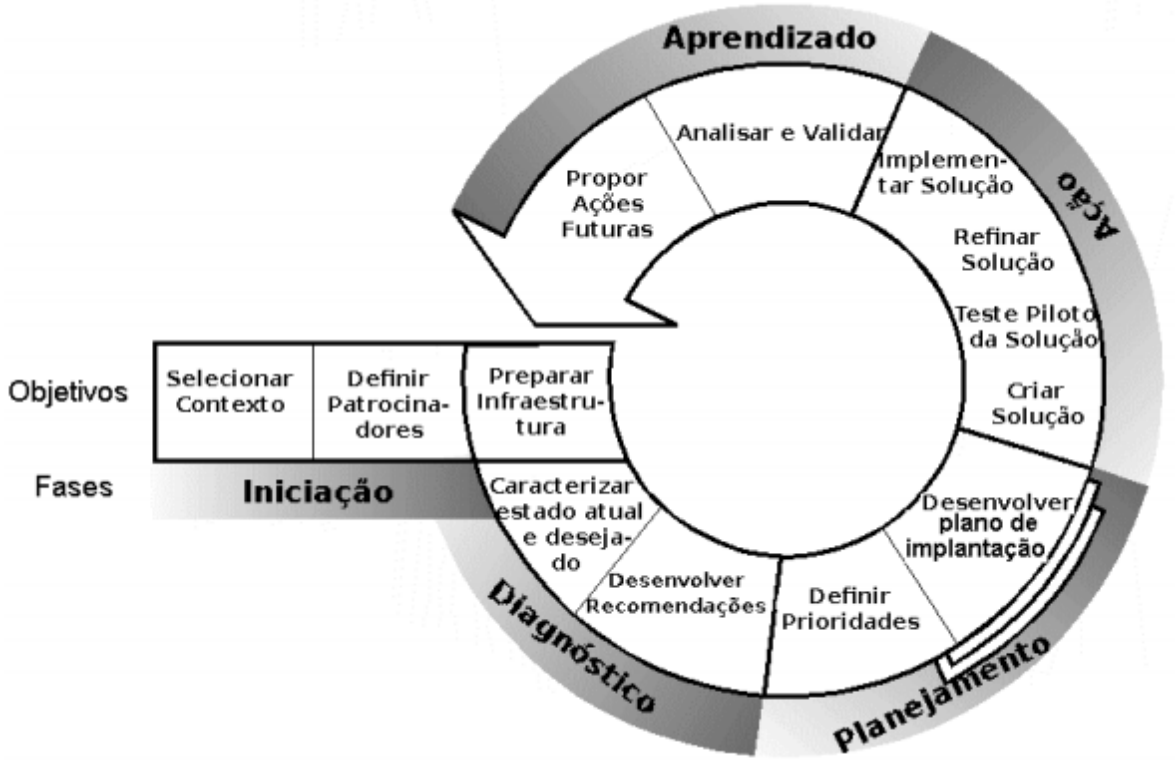


Figura 5. Ciclo de vida IDEAL (MCFEELEY, 1996 apud MENDES, 2010).

Assim como o PDCA, cada fase tem um objetivo, são eles (MCFEELEY, 1996):

- **Iniciação** - É definido principalmente o objetivo da melhoria de processo que será feita e o plano de ação é iniciado.
- **Diagnóstico** – Os dados da empresa são recolhidos e analisados e o estado atual da empresa é determinado.
- **Planejamento** – Os objetivos do SPI são priorizados e as ações são desenvolvidas, assim, o plano de ação é finalizado. Treinamento são dados caso necessário e recursos são fornecidos.
- **Ação** - Nesta fase os planos são colocados em ação, projetos pilotos são criados e feitos, e caso necessário, a solução é implantada em toda a empresa.
- **Aprendizado** – As lições aprendidas são postas em práticas e são utilizadas para melhorar o próximo ciclo do IDEAL.

2.3. QUALIDADE DE SOFTWARE

O termo Qualidade remete às características e propriedades de uma realidade, e é interpretado usualmente como o grau de conformidade de um elemento em relação a expectativa. Qualidade de Software, foi conceituada por diversos autores nas últimas décadas, de formas diversas.

Diante das necessidades relativas a contratação de serviços de TI, o tema Qualidade de Software deve ser levantado, pois fornece recursos para auxiliar nos procedimentos de Gerenciamento de Contratos na APF.

Feigenbaum (1983) afirmou que a qualidade de software seria uma determinação originada exclusivamente do cliente, e não de um engenheiro. Baseando-se então na experiência real do cliente com o produto ou serviço é medido em relação as suas exigências. Outra definição relevante em relação à qualidade de software foi descrita por Pressman (2011), que a definiu como sendo a satisfação explícita dos requisitos em termos de funcionalidade e desempenho, a utilização de padrões de desenvolvimento explicitamente documentados, e características implícitas que são esperadas em todo o software desenvolvido profissionalmente.

Entretanto, a definição do *Institute of Electrical and Eletronics Engineers* (IEEE) (1998) é a que mais se adequa no contexto deste trabalho, por se mostrar mais prática e focada na medição de qualidade. O IEEE (1998) considera que qualidade de software é o grau em que o software atende uma determinada combinação de atributos que devem ser claramente definidos. Para se avaliar os atributos definidos, um conjunto apropriado de métricas deve ser identificado. O objetivo de métricas de software é fazer avaliações ao longo do ciclo de vida do software para saber se os requisitos de qualidade de software estão sendo atendidos. O uso de métricas de software reduz a subjetividade na avaliação e controle de qualidade de software, fornecendo uma base quantitativa para a tomada de decisões sobre a qualidade do software (IEEE, 1998).

Modelos de Qualidade de Software foram desenvolvidos ao longo do tempo com o intuito de mapear a visão e as prioridades do usuário em aspectos bem definidos. Destacam-se na literatura o Modelo de Qualidade de McCall (MCCALL et al, 1977) e o Modelo de Qualidade de Boehm (BOEHM et al, 1978), os quais estão entre os principais insumos para a consolidação da ISO 9126 (AL-QUTAISH, 2010).

A ISO 9126 foi incorporada recentemente na constituição das Normas SQuaRE (ISO/IEC, 2014).

2.3.1. Normas SQuaRE

A série de normas ISO/IEC 25000 reúne um conjunto de normas conhecido como SQuaRE (Requisitos e Avaliação de Qualidade de Sistema e Software). O objetivo da série é disponibilizar um framework para avaliação da qualidade do produto de software (ISO/IEC, 2014).

Ela é o resultado da evolução de várias outras normas, especificamente a ISO/IEC 9126, a qual define um modelo de qualidade para avaliação de produto de software, e a ISO/IEC 14598, a qual define um processo para avaliação do produto de software.

Esta série de padrões da ISO/IEC 25000 é composta por 5 divisões:

- **ISO/IEC 2500n**: Os padrões que compõe esta divisão são compostos por todos os modelos, termos e definições em comum que serão utilizados por todas as outras divisões desta série.
- **ISO/IEC 2501n**: Os padrões que compõe esta divisão apresentam modelos de qualidade detalhados para sistema de computadores e produtos de software, qualidade em uso, e dados.
- **ISO/IEC 2502n**: Os padrões que compõe esta divisão incluem um modelo de referência para medição da qualidade do produto de software, definições matemáticas para medidas de qualidade, e orientação prática de suas aplicações. As medidas apresentadas são aplicadas em qualidade do produto de software e qualidade em uso.
- **ISO/IEC 2503n**: O padrão que compõe esta divisão ajuda a especificar os requisitos de qualidade. Esses requisitos podem ser utilizados no processo de elicitação de requisitos de qualidade para desenvolvimento de um produto de software ou como entrada para um processo de avaliação.
- **ISO/IEC 2504n**: O padrão é composto por requisitos, recomendações e guias para avaliação de produto de software.

A norma descreve em sua Divisão de Modelo de Qualidade (ISO/IEC 2501n) três guias principais de qualidade. Um dos modelos refere-se à Qualidade de Produto, outro a Qualidade em Uso (ambos descritos em ISO/25010) e por fim um modelo para Qualidade de Dados (ISO/IEC 25012) (ISO/IEC, 2014).

Neste trabalho será focalizado o Modelo de Qualidade de Produto que é composto de características e subcaracterísticas que se manifestam externamente no software e são resultados de atributos estáticos, que podem ser obtidos por meio de medidas internas do produto. As características especificadas no modelo são (ISO/IEC, 2014):

- **Adequação funcional**: o grau ao qual um produto ou sistema provê funções que satisfazem as especificações providas pelos usuários;
- **Eficiência de Performance**: performance relativa à quantidade de recursos usados em condições específicas;
- **Compatibilidade**: o grau no qual um produto, sistema ou componente pode trocar informações com outros produtos, sistemas ou componentes e/ou executa suas funções requeridas, enquanto compartilhando os mesmos recursos de hardware ou o mesmo ambiente;
- **Usabilidade**: o grau no qual um produto ou sistema pode ser usado por usuários específicos para atingir seus objetivos com efetividade, eficiência e satisfação em um contexto específico de uso;
- **Confiabilidade**: o grau no qual um sistema, produto ou componente executa suas funções em um contexto específico para um determinado período de tempo;
- **Segurança**: o grau no qual um produto ou sistema protege as informações e os dados de forma que pessoas ou outros produtos ou sistemas tenham o grau de acesso apropriado de acordo com níveis de autorização específicos;
- **Manutenibilidade**: o grau de efetividade e eficiência com o qual um produto ou sistema pode ser modificado pelos seus mantenedores;
- **Portabilidade**: o grau de efetividade e eficiência com o qual um sistema, produto ou componente pode ser transferido de um hardware, software ou ambiente de uso para outro;

Se as organizações proprietárias de software falham em controlar a qualidade, seus sistemas de software deterioram gradualmente ao longo do tempo e sua qualidade decai, isto afeta sobretudo a característica de Manutenibilidade (DEISSENBOECK, 2008).

2.3.2.Manutenibilidade

Manutenção de software pode ser interpretada segundo o IEEE como toda modificação de software que ocorra após sua entrega ao cliente, seja ela com intuito de corrigir falhas, melhorar características ou adaptar o produto a um ambiente modificado (IEE, 1993). Estima-se ainda que a manutenção de software represente cerca de 80% do ciclo de vida do software (PIGOSKI, 1996), corroborando sua relevância e necessidade de atenção durante o desenvolvimento de software.

A manutenção de produtos de software pode-se apresentar sob três tipos principais: reparativas (também conhecidas como corretivas), adaptativas, e perfectivas (SOMMEVILLE, 2011).

Manutenibilidade, por sua vez, caracteriza-se como a facilidade, precisão, segurança e economia na execução de ações de manutenção inerentes a um sistema ou produto (BLANCHARD, 1992).

A SQuaRE define Manutenibilidade como sendo o grau de eficácia e eficiência com o qual um produto ou sistema pode ser modificado por determinado mantenedor. (ISO/IEC, 2014). Manutenibilidade é classificada como sendo uma Característica de Qualidade de acordo com o Modelo de Qualidade de Produto contido na norma. Suas subcaracterísticas são (ISO/IEC, 2014):

- **Modularidade:** grau em que um software é composto por componentes independentes, de tal modo que uma alteração em um componente tem um impacto mínimo sobre os outros componentes.
- **Reusabilidade:** grau em que um ativo pode ser usado em mais de um sistema, ou na construção de outros ativos.
- **Analisabilidade:** grau de eficácia e eficiência com que é possível avaliar o impacto de mudança destinada a um ou mais dos seus componentes.

- **Modificabilidade:** grau em que um software pode ser eficaz e eficientemente modificado sem a introdução de defeitos ou degradação da sua qualidade.
- **Testabilidade** grau de eficácia e eficiência com a qual é possível estabelecer critérios de teste para um software ou componente e a realização dos testes para determinar se esses critérios foram cumpridos.

Sommerville (2011) classifica os fatores técnicos que afetam a manutenção são eles: independência entre módulos, linguagem de programação, estilo de programação, teste e validação de produtos de software, qualidade da documentação do produto de software e técnicas de gerência de configuração.

A literatura faz referência a relação de diversas particularidades de produto com Manutenibilidade de Software. Entre elas estão as métricas de complexidade de produto, com destaque para Complexidade Ciclomática (Mills, 1988), aspectos arquiteturais (SEI, 2005) e design de software (ROMBACH, 1990).

Sobre cobertura do código por testes unitários alguns autores acreditam que testes automatizados podem prejudicar a manutenibilidade pois costumam consumir muito esforço para alteração (FIELDS, 2010). Em contrapartida, outros afirmam que tornando o código mais estável em termos de funcionalidade é evitada a inserção de comportamentos indesejados em outros pontos do sistema, além de que testes automatizados também são considerados documentação e ajudam o entendimento do código, consequentemente a manutenção (KUMAR, 2014).

2.3.3. Qualidade de Código

Qualidade de código é a medida do quão adequado um determinado código-fonte está em relação ao desenho (arquitetura, por exemplo) e as regras de codificação pré-definidas (MCCONNELL, 2004). Esta categoria de qualidade pode ser medida utilizando a relação entre números de elementos com defeitos e o número total de elementos (FENTON, 1997).

Diferente da Qualidade de Produto, que é uma característica visível em um sistema de software, qualidade de código está escondida e pode vir a se tornar visível muito depois que o projeto é finalizado (JIANG, MENZIES, 2008). Entretanto, as consequências da baixa qualidade de código são muito maiores (MISRA, 2005).

Com a adoção de metodologias ágeis pela indústria de software, o código-fonte se tornou um dos artefatos mais importantes para se medir a qualidade do software (BENKLER, 2006). Com isso, as métricas de qualidade de código são um mecanismo importante para a avaliação desses sistemas.

A utilização de métricas de qualidade de código-fonte como critério para a avaliação da qualidade de um software é motivada por estudos (SELLERS, 1996) (SATO, 2007) que indicam ser viável analisar algumas das principais características para a aceitação de um software, tais como: flexibilidade, complexidade e manutenibilidade a partir do código-fonte da aplicação. Além disto, há outros motivos para monitoramento da qualidade de código, por exemplo, desenvolvedores tomam decisões sobre o código enquanto estão programando; estas decisões influenciam fortemente a qualidade do código (BECK, 2007).

As ferramentas de avaliação de qualidade de código normalmente apresentam métricas, ou seja, valores numéricos, para que se possa interpretar se a qualidade de código-fonte está boa ou ruim. Há várias métricas para avaliação do código-fonte, a maioria baseada na análise estática do código permitindo assim a avaliação do produto de software (GOUSIOS et al., 2007).

Há várias métricas de código-fonte; elas podem ser divididas em métricas de tamanho, de complexidade, de manutenibilidade e outras (MEIRELLES, 2013). Podemos destacar algumas de cada tipo:

A. Métricas de tamanho

Estas métricas foram feitas para tentar quantificar o tamanho de um software. As duas principais, ou mais utilizadas seriam:

- **Linhas de Código (LOC)** - Parte do princípio de contabilizar quaisquer linhas que não seja em branco ou comentário, independentemente do número de declarações por linha (BOEHM, 1981).
- **Pontos de Função (PF)** - Baseado nos modelos de ciclo de vida clássicos de software. A métrica calcula o valor total Pontos de Função para o projeto, que tem como parâmetros o número de entradas do usuário, as consultas, as saídas e os principais arquivos (ALBRECHT, GAFFNEY, 1983).
- **Métricas de Halstead** - É um conjunto de métricas baseadas na teoria da informação, consideradas as primeiras métricas com fundamentação comum

(HALSTEAD, 1972). Essas métricas se aplicam à vários aspectos do software e seu esforço para desenvolvimento. Comprimento (N) e Volume (V) são métricas que se aplicam especificamente ao software final. Também há métricas para esforço total (E) e tempo de desenvolvimento (T).

B. Métricas de Complexidade

Estas métricas são utilizadas para medir de forma objetiva a complexidade de uma parte do software. As principais são:

- **Complexidade Ciclomática** - Parte do princípio que a complexidade depende do número de condições (caminhos), correspondendo ao número máximo de percursos linearmente independentes de um software (MCCABE, 1976).
- **Fluxo de Informação** - Foi proposto que o fluxo de informação é uma medida de complexidade. Esse método conta o número de parâmetros de entrada (fan-in) e saídas (fan-out) (KAFURA, HENRY, 1981).

C. Métricas de Manutenibilidade

O SEI recomendo o uso de um índice de manutenibilidade. A manutenibilidade de um sistema é calculada usando uma combinação de resultados de outras métricas (VANDOREN, 1997). O índice tem a seguinte fórmula:

$$171 - 5{,}2\ln(aveV) - 0{,}23aveV(G') - 16{,}2 * \ln(aveLOC) \\ - 50\sin(\sqrt{2{,}4 * perCM})$$

Os coeficientes sugeridos são resultados de calibragem usando vários sistemas mantidos pela HP (Hewlett-Packard). Os termos são os seguintes:

- aveV é a média do volume de Halstead;
- aveV(G) é a média da complexidade ciclomática por módulo;
- aveLOC é a média das linhas de código por módulo;
- perCM(opcional) é a média da porcentagem de linhas de comentário por módulo;

D. Outras Métricas

Existem ainda outras métricas. Algumas são parecidas com as anteriores citadas, outras utilizam de algumas das anteriores para formularem métricas mais complexas, enquanto outras são conceitos totalmente novos. Aqui serão citadas algumas:

- **Razão de Coesão** - Mede a relação entre o número de módulos com coesão funcional e o número total de coesão (BIEMAN, OTT, 1994).
- **Tamanho Médio dos Módulos** - Mede o tamanho médio dos módulos que compõe o programa (BOEHM, 1978).
- **Número de Funções** - Mede o número de funções e as linhas de código das funções (SMITH, 1980).
- **Relação de Acoplamento** - Sinaliza uma relação de todo par de módulos segundo o tipo de acoplamento (FENTON, MELTON, 1990).
- **Modularidade Global** - Descrição da modularidade global em termos de várias visões específicas de modularidade (HAUSEN, 1989).

Estas são algumas das métricas de qualidade de código existentes. Vale lembrar que, dependendo da ferramenta sendo utilizada, os cálculos podem variar, mas a base teórica, principalmente, irá se manter.

Atualmente, há várias ferramentas que avaliam essa qualidade baseada nas regras pré-definidas citadas, entre as ferramentas podem ser lembradas PMD, *FindBugs*, *CheckStyle* e *SonarQube*. Como dito, estas ferramentas têm como foco verificar o software e encontrar problemas em nível de código.

2.4. VERIFICAÇÃO DE SOFTWARE

Os processos de Verificação e Validação de software tem uma estreita relação com a qualidade do produto. É através deles que engenheiros de software buscam qualidade, por meio da aplicação de métodos e medidas técnicas sólidas, conduzindo revisões técnicas formais e efetuando teste de software bem planejado (PRESSMAN, 2011).

Verificação é “confirmar que cada serviço e/ou produto de trabalho do processo ou do projeto reflete apropriadamente os requisitos especificados” (SOFTEX, 2012). Validação é “confirmar que um produto ou componente do produto atenderá a seu uso

pretendido quando colocado no ambiente para o qual foi desenvolvido" (SOFTEX, 2012).

Verificação e validação incluem uma grande quantidade de atividades de GQA (Garantia da Qualidade): revisões técnicas, inspeção de software, auditorias de qualidade e configuração, revisão de base de dados, análise de algoritmo, teste de desenvolvimento, teste de usabilidade, teste de aceitação e teste de instalação (PRESSMAN, 2011).

Este trabalho tem como foco o processo de Verificação de Software. É durante o processo de verificação que é possível determinar se os produtos de software de uma atividade atendem completamente aos requisitos ou condições impostas a eles nas atividades anteriores (NBR ISO/IEC, 2003). Ou seja, o processo verificação de software faz parte das atividades de monitoramento da execução do contrato de contratação, principalmente na avaliação da qualidade dos bens entregues e avaliação dos critérios de aceitação definidos em contrato de acordo com a Instrução Normativa MP/SLTI Nº04 (BRASIL, 2014a).

A inspeção de Software é um meio formal de verificação de atributos de qualidade, principalmente em relação a conformidade e manutenção (SOMMERVILLE, 2011), por este motivo deve ser visto mais a fundo neste trabalho.

2.4.1.Inspeção de Software

Inspeção de Software é um processo de verificação e validação estático, no qual um sistema de software é revisto para se encontrar erros, omissões ou anomalias. É uma revisão formal de software, sendo seu principal objetivo a descoberta antecipada de falhas (FELIZARDO, 2004).

Inspeções focam no código-fonte, mas também é possível utilizá-las para avaliar requisitos ou modelos de projeto. Há três principais vantagens na sua utilização (SOMMERVILLE, 2011):

- Não há necessidade de se preocupar com interação entre erros do sistema. Uma única sessão de inspeção pode descobrir muitos erros de um sistema;
- Versões incompletas de um sistema podem ser inspecionadas;
- Uma inspeção pode considerar atributos de qualidade mais amplos como conformidade com padrões e manutenção.

As inspeções não possuem apenas vantagens, elas são dispendiosas e só ocorre economia de custos depois que a equipe de desenvolvimento se torna experiente em seu uso (SOMMERVILLE, 2011). Em contrapartida, cerca de 60% dos defeitos encontrados em software podem ser capturados pelas inspeções (BOEHM; BASILI, 2001).

Inspeções são uma forma de análise estática dirigidas por checklists de erros e heurísticas que identificam erros comuns em diferentes linguagens de programação. Este checklist pode ser baseado em exemplos de livros ou de conhecimento de defeitos comuns em determinado domínio da aplicação. Checklists devem ser diferentes para linguagens diferentes, pois cada linguagem tem seus erros característicos (SOMMERVILLE, 2011).

Métricas derivadas do código em geral tornam inviáveis verificações manuais, pois o fator humano pode comprometer a confiabilidade dos valores obtidos, além de ser financeiramente impraticável. A opção mais adequada para métricas de código são as verificações automáticas, onde destaca-se a Análise Estática Automatizada.

2.4.2. Verificação Automática

Para alguns erros e heurísticas, é possível automatizar o processo de verificação de programas, o que culminou no desenvolvimento de analisadores estáticos automatizados para diferentes linguagens de programação (SOMMERVILLE, 2011).

Estes analisadores são ferramentas de software que varrem o código-fonte de um programa detectando defeitos e anomalias. Pela sua construção, podem detectar se as declarações estão bem formuladas, até fazer inferência sobre o fluxo de controle e extrair um conjunto de todos os valores possíveis para os dados do programa. Normalmente, eles suprem os recursos de detecção de erros que o compilador não provê (SOMMERVILLE, 2011).

Mas apenas os analisadores não podem garantir que o código está completamente adequado, ou que ele não possui nenhum falso positivo. Principalmente porque eles, os analisadores, são baseados na ideia de que o erro humano é previsível e que, os próprios programadores definem quais regras serão

usadas e, se alguma em especial for necessária, eles podem escrever regras customizadas ao longo do processo (LOURIDAS, 2006).

Estas regras customizadas podem ser escritas para as ferramentas analisadoras já existentes. Um exemplo de ferramenta é o *SonarQube*, que é uma plataforma aberta para controle de qualidade de código (SONAR, 2014). Ele é basicamente uma ferramenta web baseada em *plugins*, que manipula os resultados de várias ferramentas de análise de código (STORCH; LAUE; GRUHN, 2013). Neste trabalho o *SonarQube* foi escolhido dentre outras ferramentas semelhantes por ser uma ferramenta com grande difusão no mercado, além de ser a plataforma onde o Perfil de Qualidade para Órgãos da APF foi customizado (CROZARA, 2014), abordaremos a aplicação do perfil mais a frente, na Seção 3.2.3.

O princípio de funcionamento do *SonarQube* pode ser descrito da seguinte forma: Depois de configurar um projeto para que ele passe a atuar, os arquivos de códigos-fonte no projeto serão examinados pelos vários *plugins* existentes, que foram definidos na configuração. Após examinados, os dados gerados por cada *plugin* são disponibilizados para o *SonarQube* em forma de "*reports*". Assim, a ferramenta gera várias visualizações que integram os resultados obtidos pelos *plugins* (STORCH; LAUE; GRUHN, 2013).

Por exemplo, a visualização da *DashBoard* mostra arquivos ou pacotes com maiores problemas, as métricas globais analisadas no projeto, como cobertura de código, a quantidade de quebras de regras de codificação existentes e sua gravidade. *SonarQube* é totalmente extensível, há muitos *plugins* que podem ser adicionados para dar suporte à análise de uma linguagem. Diferentes perfis de qualidade podem ser utilizados para definir o escopo dos resultados gerados pelos reports. Utilizando um protocolo para o processo de análise, timelines do projeto podem ser criadas. Assim, o progresso da qualidade de um projeto pode ser medido (HASHIURA; MATSUURA; KOMIYA, 2010).

O *SonarQube* possui suporte para mais de 20 linguagens de programação, entre elas: Android, C/C++, Java, Groovy, PHP, Python e COBOL (SONAR, 2014). E analisa basicamente sete áreas de qualidade de código, são elas:

- Arquitetura e Design
- Duplicações
- Testes
- Complexidade

- Bugs em potencial
- Regras de codificação
- Comentários

Devido às características da implantação que ocorrerá no Órgão X, onde as entregas necessitam de velocidade, decorrente da adoção de metodologias ágeis de desenvolvimento, os resultados também precisam ser apresentados com velocidade e precisão, além de que este trabalho está centrado em uma característica de qualidade específica, a manutenibilidade, apenas três destas áreas do *SonarQube* vão ser exploradas: Complexidade, Arquitetura e Design e Testes.

A Complexidade citada entre as áreas de qualidade analisadas é a Complexidade Ciclomática de McCabe (1976). No *SonarQube*, ela é calculada utilizando as palavras chaves presentes no código como *if, else, while, for, return*. Para cada palavra chave é adicionado à complexidade da função o valor de +1, assim como laços de repetição que são contabilizados pela quantidade de repetições a serem executadas. Qualquer método ou função sempre terá no mínimo complexidade igual à 1 por causa de sua declaração *public void, public int* entre outras. A complexidade da classe é dada pelo número total de complexidade no arquivo dividido pelo número de métodos (SONAR, 2014).

Arquitetura e Design no *SonarQube* se refere em maior parte às dependências entre arquivos dentro dos diretórios, interdependências entre diretórios e dependências não desejadas. Algumas delas estão em outros níveis como de módulo e de projeto, além de diretórios. Essa dependência vai, no fim, referir-se ao acoplamento entre as classes de projeto, que idealmente deve ser o mais baixo quanto possível, concomitantemente a uma coesão a mais alta possível (LOURIDAS, 2006).

Finalmente, Cobertura de Testes no *SonarQube* é derivada das métricas de cobertura de linha e cobertura de *branch*, e responde apenas uma questão: “Quanto do código-fonte foi coberto pelos testes unitários? ”. A resposta é dada através de porcentagem, onde quanto mais perto de 100% mais bem coberto o código está (SONAR, 2014).

2.5. INTEGRAÇÃO CONTÍNUA

Integração Contínua é uma prática de desenvolvimento de software onde membros da equipe incorporam mudanças ao software frequentemente, utilizando processos de compilação e testes que asseguram a integridade do projeto (FOWLER, 2006).

Segundo Durvall Paul um dos primeiros autores a utilizar o termo Integração Contínua foi Grady Booch, que atribuiu como sendo “um processo realizado em intervalos regulares, que resulta em um executável que incorpora o aumento funcional do sistema”, além disso ele sugere que este processo seria uma espécie de “marco de projeto onde a gerência ´poderia realizar medições para controlar melhor o projeto e atacar os riscos em uma base contínua” (PAUL, 2007).

Com o advento das Metodologias Ágeis diversas técnicas e práticas que permitem dinamizar o desenvolvimento de software ganharam destaque. O conceito de Integração Contínua foi aprimorado (PAUL, 2007), passando a ser recomendado não apenas em intervalos regulares, mas sempre que possível, por viabilizar um ambiente coeso e propício para redução significativa de problemas de integração (XP, 1999), além de atenuar uma das características mais críticas de software, a invisibilidade (FOWLER, 2006).

A Integração Contínua é um passo inicial para uma outra prática fortemente encorajada, a Entrega Contínua (*Continuous Delivery*) (THOUGHTWORKS, 2015), que se trata de uma disciplina de desenvolvimento de software onde o sistema é construído de tal forma que pode ser liberado para produção em qualquer momento (FOWLER, 2015). Os principais benefícios de entregar continuamente são reduzir risco de *deploy*, melhoria na confiança do progresso do projeto (aumento de visibilidade) e adiantamento do *feedback* do usuário (antecipação de mudanças) (FOWLER, 2015).

Atualmente diversas ferramentas promovem integração contínua identificando mudanças no repositório do projeto, verificando o código e executando um conjunto de procedimentos para verificar se a mudança é boa e não irá prejudicar alguma parte do projeto. Estas ferramentas atuam como um “juiz imparcial” (MEYER, 2014), e em geral elas atuam conforme mostra a Figura 6.

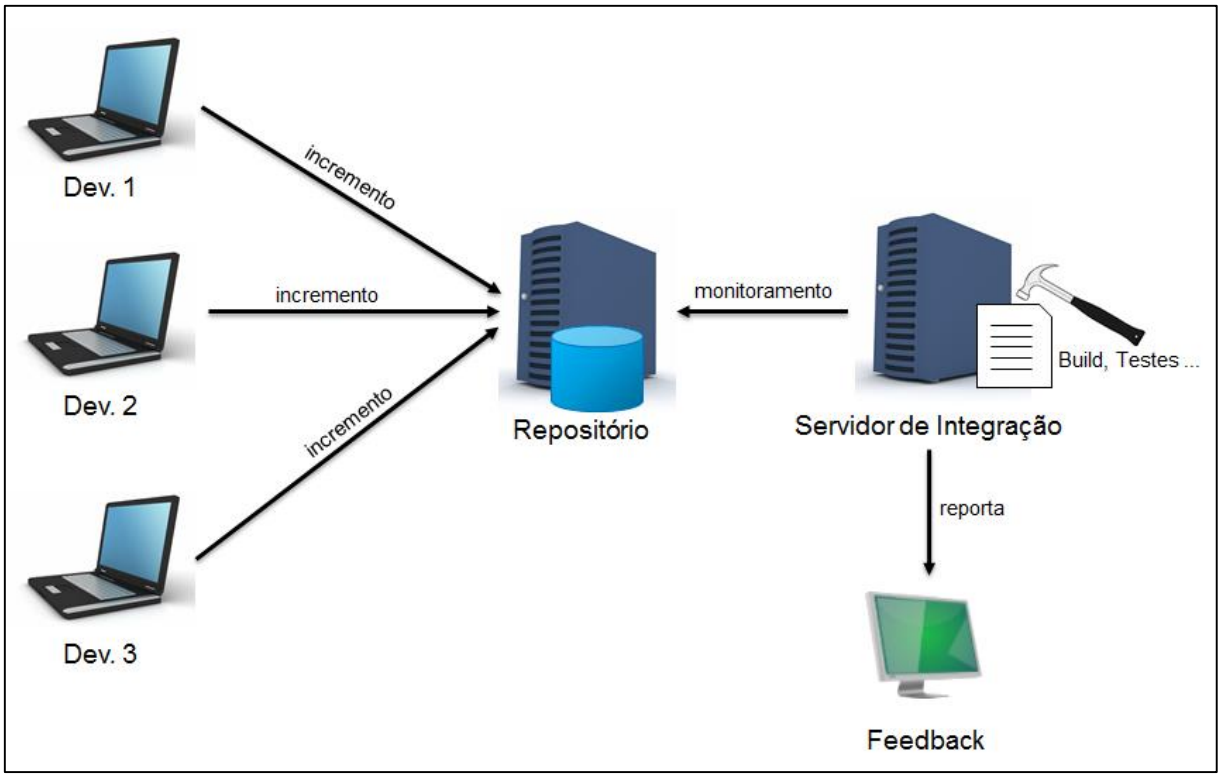


Figura 6. Arquitetura de Integração Contínua (Adaptado, MOREIRA 2015)

Algumas ferramentas bastante difundidas para integração contínua são "*Cruise Control*", "*Continuum*", "*Team City*", "*Hudson*" e "*Jenkins*". Entre elas o *Jenkins* conseguiu maior alcance na comunidade *open-source*, consequentemente tem vantagem na identificação e correção de bugs, implementações de melhorias, bem como no desenvolvimento de *plugins* compatíveis (SMART, 2011).

A ferramenta *Jenkins* é originária do mesmo projeto da ferramenta Hudson, cuja teve sua primeira versão disponibilizada em 2005. Alguns impasses relacionados aos direitos e decisões de projeto envolvendo a empresa Oracle culminaram na "separação" dos projetos e adoção do nome *Jenkins* em 2011 (JENKINS, 2011).

Jenkins é um software livre e *open-source* para integração contínua, desenvolvido em Java. Além de monitorar, integrar (através de compilação e testes) e fornecer *feedback* sobre os projetos, ele permite que sejam customizados diversos procedimentos adicionais através de *plugins* como por exemplo o "*SonarQube plugin*", que permite a execução remota de inspeções de qualidade de código utilizando o Sonar (JENKINS, 2015).

3. ESTUDO DE CASO

O objeto deste estudo de caso é um órgão da APF, denominado neste trabalho de **Órgão X**. Este trabalho faz parte de um Projeto de Melhoria de Gestão de Demandas de Software neste órgão. O projeto é oriundo de um Termo de Cooperação entre a Universidade de Brasília (UnB) e o órgão em questão, cuja intenção primária é atender as necessidades de melhoria de TI no órgão. Entre as principais iniciativas decorrentes do projeto está a adoção de metodologias ágeis na definição de um processo de gestão de demandas de desenvolvimento de software por terceiros, empregando o *Scrum* e sua perspectiva de *Sprint*, visando entregas contínuas, que sempre contenham valor de produto para o cliente.

Considerando o objetivo primário deste estudo, “elaborar uma estratégia de verificação de qualidade adequada para o Órgão X”, foram estabelecidos alguns procedimentos. Primeiramente será apresentado um **diagnóstico** sobre o órgão destacando qual a situação atual quanto qualidade de software e serão apontados potenciais problemas.

Em seguida será definida a **estratégia de qualidade** de forma a tratar os problemas identificados, considerando os aspectos de métricas, ferramentas, papéis e procedimentos.

Posteriormente a estratégia de qualidade será testada e os resultados obtidos serão analisados, **validando** assim a solução.

3.1. DIAGNÓSTICO

Em “Caracterização do Órgão X” (3.1.1), são apresentados o contexto geral do órgão quanto a área de TI. Em “Verificação de Qualidade no Órgão X” (3.1.2) é apresentada a situação atual da qualidade de software na instituição abordando os procedimentos adotados e a situação atual dos sistemas legados. E por fim, em “Oportunidades de Melhoria” (3.1.3) são identificados os problemas relacionados a qualidade de software, a partir da análise da conjuntura atual do Órgão.

3.1.1. Caracterização do Órgão X

As áreas de jurisdição do Órgão X abrangem serviços de radiodifusão, postais e de telecomunicações. Os objetivos estratégicos do órgão em relação à TI foram estabelecidos no Plano Diretor de TI (PDTI) para o período de 2013 a 2015 (BRASIL, 2014b) e podem ser vistos na Tabela 2.

Tabela 2 - Objetivos estratégicos PETI. (BRASIL, 2014b, adaptado.)

OE-PETI 1	Promover a governança de TI no Órgão X.
OE-PETI 2	Evoluir no atendimento das áreas finalísticas do Órgão X, balanceando o portfólio de projetos e serviços a partir das diretrizes do planejamento estratégico do ministério.
OE-PETI 3	Aprimorar a gestão de TI no Órgão X.
OE-PETI 4	Redefinir a estrutura organizacional e a composição das equipes envolvidas nas atividades de TI do Órgão X.
OE-PETI 5	Melhorar continuamente os serviços de TI no Órgão X.
OE-PETI 6	Estabelecer e aprimorar a (s) arquitetura (s) de referência para a TI do Órgão X, de forma alinhada com as diretrizes do governo federal.
OE-PETI 7	Prover soluções confiáveis e disponíveis que ofereçam recursos de: mobilidade, colaboração, desmaterialização de processos e transparência para com a sociedade.
OE-PETI 8	Prover a segurança da informação e comunicação no Órgão X.

Com o intuito de atender, particularmente aos Objetivos Estratégicos 3 e 5, foi proposto um projeto de melhoria de processos, o qual inclui um processo Ágil de Gestão de Demandas de Software, denominado GeDDAS, cujo caracteriza-se como “plano de fundo” para a Estratégia de Verificação de Qualidade de Software a ser proposta neste trabalho.

O OE-PETI 3 (Tabela 2) aborda a importância em otimizar a gestão de TI no Órgão X. Conforme elucidado na seção 2.1.1 a gestão de TI envolve, entre outros fatores, o acompanhamento da solução contratada, incluindo avaliação de qualidade de produto de software. O propósito desta pesquisa está, portanto, reforçado considerando este objetivo estratégico.

No OE-PETI 5 (Tabela 2) é dada ênfase a necessidade de melhorar continuamente os serviços de TI no Órgão X. O propósito do Projeto de Melhoria de

Gestão de Software, colabora para o alcance do objetivo proposto, além de evidentemente convergir no sentido de atingir o objetivo estratégico OE-PETI 3.

O GeDDAS é advindo de um trabalho realizado por Souza et al (2015) e busca promover uma perspectiva ágil na gestão de demandas de software no Órgão X, orientado pelos princípios do *Scrum*. O processo idealizado por Souza et al (2015) sofreu algumas adaptações e o modelo conforme adotado pelo órgão pode ser visto na Figura 7.

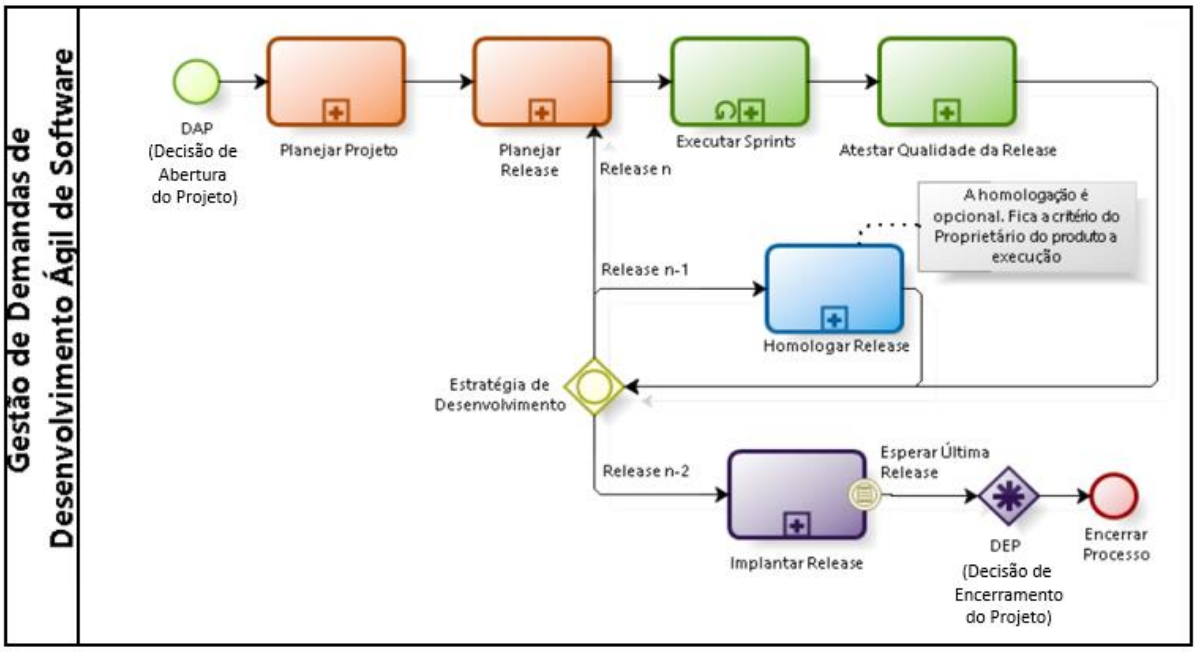


Figura 7- GeDDAS. (Brasil, 2014e)

3.1.2. Qualidade no Órgão X

O Órgão X atualmente conta com uma empresa terceirizada para Apoio ao Controle de Qualidade, cujo escopo dos serviços prestados envolvem (BRASIL, 2012a):

I. Aferição dos resultados, verificação do cumprimento dos níveis de serviço acordados e metrificação dos pontos de função das atividades de desenvolvimento e manutenção de sistemas;

II. Aferição dos resultados e verificação do cumprimento dos níveis de serviços acordados para as atividades de Sustentação do Ambiente de Tecnologia da Informação e Comunicações (TIC);

III. Aferição dos resultados e verificação do cumprimento dos níveis de serviço acordados para as atividades de apoio à gestão de TIC;

IV. Relatórios de progresso das atividades contratadas;

V. Apoio à gestão de projetos, análise e modelagem de processos e, governança de TIC.

As condições para a execução destes serviços são disciplinadas pelo Termo de Referência do órgão, o qual trata-se de um documento elaborado com o objetivo de definir as condições para a contratação da prestação de serviços e soluções técnicas na área de Tecnologia da Informação e Comunicação (BRASIL, 2011b).

Até então, em termos práticos, o Órgão X juntamente com a empresa contratada para Apoio ao Controle de Qualidade trabalham baseados em alguns indicadores na prestação de serviços de manutenção e desenvolvimento de software, são eles:

- **Sucesso de Prazo de Fase**- diz respeito a pontualidade no encerramento das fases no desenvolvimento de software.
- **Sucesso de Prazo Global**- refere-se à pontualidade do projeto como um todo, levando em consideração o prazo para finalização do projeto e o prazo real executado.
- **Qualidade dos Artefatos**- avaliação dos artefatos produzidos em uma entrega parcial ou total, identificando não conformidades proporcionalmente ao tamanho funcional da solução. Para a composição do indicador de Qualidade dos Artefatos são utilizados *checklists* com questões relacionadas a boas práticas e conteúdo esperado nos artefatos.
- **Qualidade Global do Serviço**- trata sobre funcionamento inadequado ou não conforme aos requisitos, sobretudo funcionais, do sistema.

O novo processo de gestão de demandas (GeDDAS), prevê procedimentos relacionados a verificação de qualidade, e estabelece o termo “Conceito de Pronto”, como sendo um conjunto de requisitos variáveis para um incremento de software ser considerado conforme e de qualidade.

O Sub-Processo Atestar Qualidade da *Release* prevê, inclusive, a atividades “Atestar Qualidade do Incremento de *Software*”, a qual tem como objetivo garantir que o incremento de software possua qualidade suficiente para ser implantado em

ambiente de homologação. É sugerido em um dos procedimentos da atividade que o fornecedor apresente métricas de qualidade de código.

O trabalho de Crozara (2014) realizou uma avaliação da qualidade de código dos principais sistemas legados do órgão e revelou o estado de degradação em que se encontram. A autora customizou as regras da ferramenta de análise estática *SonarQube* conforme a necessidade do órgão e atribuiu severidades, conforme pode ser visto na Tabela 3.

Tabela 3- Classificação das Violações (Adaptado, CROZARA 2014)

Severidade	**Tipos de Violações**
Muito Alta	Segurança e vulnerabilidade do código.
Alta	Más práticas de programação, corretude, ineficiências, bugs, bugs em potencial e indicativos de erros de programação.
Média	Más práticas de programação, corretude, ineficiências, futuros erros de programação, inconsistências de estilo, bugs e bugs em potencial.
Baixa	Inconsistências de estilo, corretude, ineficiências e indicativos de erro de programação.
Muito Baixa	Inconsistências de estilo.

A distribuição das violações nos projetos avaliados em relação às severidades se dá, em média, da seguinte forma: 1% são violações são de severidade “Alta”, 25% são “Médias”, 72% são “Baixas”, e 2% são consideradas “Muito Baixas”. Nenhuma violação “Muito Alta” foi encontrada nos projetos avaliados, na Figura 8.

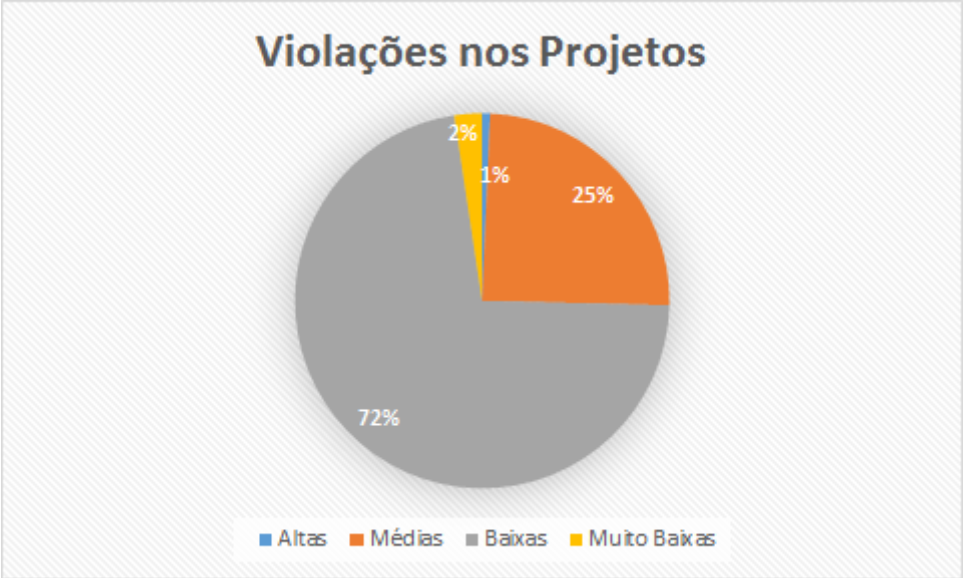
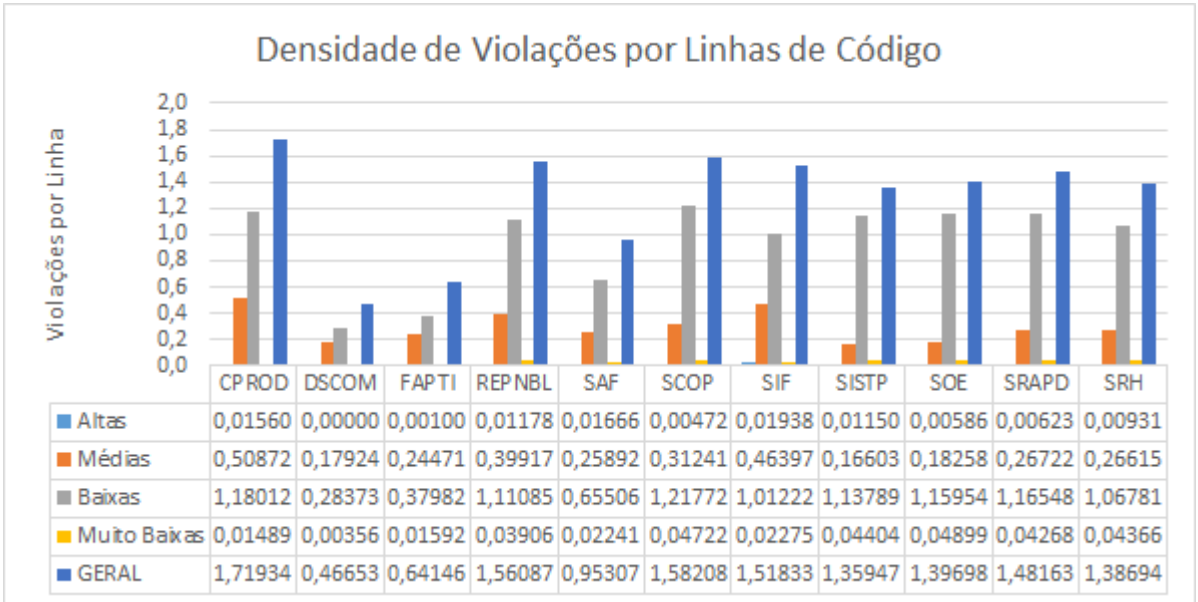


Figura 8- Distribuição de Severidades nos Projetos (Adaptado, CROZARA, 2014)

O fator mais preocupante, porém, é a quantidade de ocorrências encontradas quando observada a quantidade de linhas de código executáveis. Foram encontradas 219.521 violações (de todas as severidades) em todos os projetos, que somam 182.427 linhas de código, ou seja, a densidade de violações no geral representa mais de 1 violação por linha de código executável (cerca de 1,2 violações por linha), as densidades detalhadas podem ser vistas na Figura 9.



	CPROD	DSCOM	FAPTI	REPNBL	SAF	SCOP	SIF	SISTP	SOE	SRAPD	SRH
Altas	0,01560	0,00000	0,00100	0,01178	0,01666	0,00472	0,01938	0,01150	0,00586	0,00623	0,00931
Médias	0,50872	0,17924	0,24471	0,39917	0,25892	0,31241	0,46397	0,16603	0,18258	0,26722	0,26615
Baixas	1,18012	0,28373	0,37982	1,11085	0,65506	1,21772	1,01222	1,13789	1,15954	1,16548	1,06781
Muito Baixas	0,01489	0,00356	0,01592	0,03906	0,02241	0,04722	0,02275	0,04404	0,04899	0,04268	0,04366
GERAL	1,71934	0,46653	0,64146	1,56087	0,95307	1,58208	1,51833	1,35947	1,39698	1,48163	1,38694

Figura 9- Densidade de Violações por Linhas de Código (Adaptado, CROZARA, 2014)

3.1.3.Oportunidades de Melhoria

Dado o contexto do Órgão X, foram levantadas as causas do principal problema identificado (degradação dos sistemas legados), que para este trabalho, são oportunidades de melhoria.

As causas identificadas são das áreas específicas: Processos, referente as atividades realizadas ou não em relação a qualidade de código; Ambiente, sobre recursos existentes, não existentes ou mal utilizados; e Critérios, referente a como as métricas de qualidade de código estão sendo utilizadas a favor da melhoria do projeto. O diagrama de causa e efeito a seguir mostra hipóteses de causas que colaboram para a existência do problema na Figura 10.

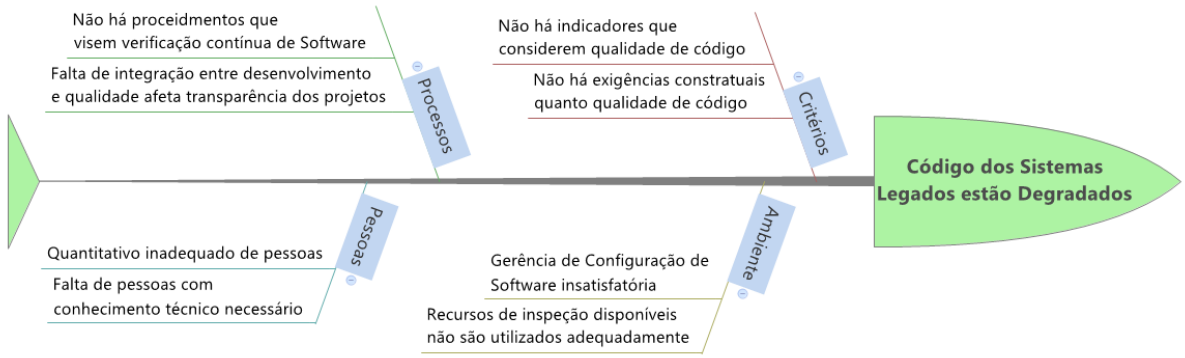


Figura 10- Diagrama de Causa e Efeito

As causas foram priorizadas considerando o contexto burocrático em que as instituições da APF estão inseridas, portanto foram priorizadas as causas que pudessem ser solucionadas de forma não dispendiosa para o Órgão X.

Tabela 4- Relação entre causas e ações previstas.

	Causa a ser Tratada	Ação Prevista na Estratégia
Critérios	Não há indicadores que considerem qualidade de código	Definição de Métricas e Indicadores de Qualidade de Código.
	Não há exigências contratuais quanto qualidade de código	Não se aplica.
Ambiente	Gerência de Configuração de Software Insatisfatória	Não se aplica.

	Recursos de inspeção disponíveis não são utilizados adequadamente	Seleção e implantação de ferramentas de análise estática e integração contínua.
Processos	Não há procedimentos que visem verificação contínua de software	Definição dos procedimentos de verificação de qualidade de código.
	Falta de integração entre desenvolvimento e qualidade afeta a transparência dos projetos	Aumento da visibilidade através de ferramentas.
Pessoas	Quantitativo inadequado de pessoas	Não se aplica.
	Falta de pessoas com conhecimento técnico necessário	Não se aplica.

3.2. ESTRATÉGIA DE VERIFICAÇÃO DE QUALIDADE DE CÓDIGO

Nesta seção serão descritas as definições para a execução da estratégia de verificação de qualidade, as métricas a serem analisadas, as ferramentas, os papéis e suas responsabilidades e os procedimentos para execução.

3.2.1.Métricas

3.2.1.1. Seleção de Métricas

Foram selecionados três domínios principais a serem observadas no código e que impactam diretamente a qualidade de código dos sistemas: **Complexidade de Software**, através da aferição de Complexidade Ciclomática; **Testes de Software**, observando a Cobertura de Testes; **Práticas de Programação**, aplicando o Perfil de Qualidade para Órgãos da APF (Crozara, 2014).

Complexidade Ciclomática e Cobertura de testes são métricas já difundidas no âmbito de análises de código e neste estudo serão medidas por meio de ferramenta de análise estática, enquanto as práticas de programação serão verificadas conforme definido no trabalho de Crozara (2014).

As práticas de programação a serem analisadas serão as determinadas no Perfil de Qualidade para Órgãos da Administração Pública Federal desenvolvido no

trabalho de Crozara (2014). No trabalho referido a autora realizou a customização da ferramenta *SonarQube* utilizando os *plugins FindBugs*, *Squid*, PMD e *Checkstyle*, em seguida promoveu a identificação das regras repetidas entre as ferramentas, identificou as violações de regras mais recorrentes, realizou uma análise aprofundada segundo impacto, severidade e classificação. Por fim, no trabalho de Crozara (2014) foram mapeadas as sub-características de qualidade obtendo-se assim o Perfil de Qualidade para Órgãos da APF (Anexo 1).

Um esquema para elucidar sobre as relações entre as métricas selecionas e métricas derivadas pode ser visto na Figura 11. Em seguida, na Tabela 5, é apresentado o detalhamento das métricas selecionadas.

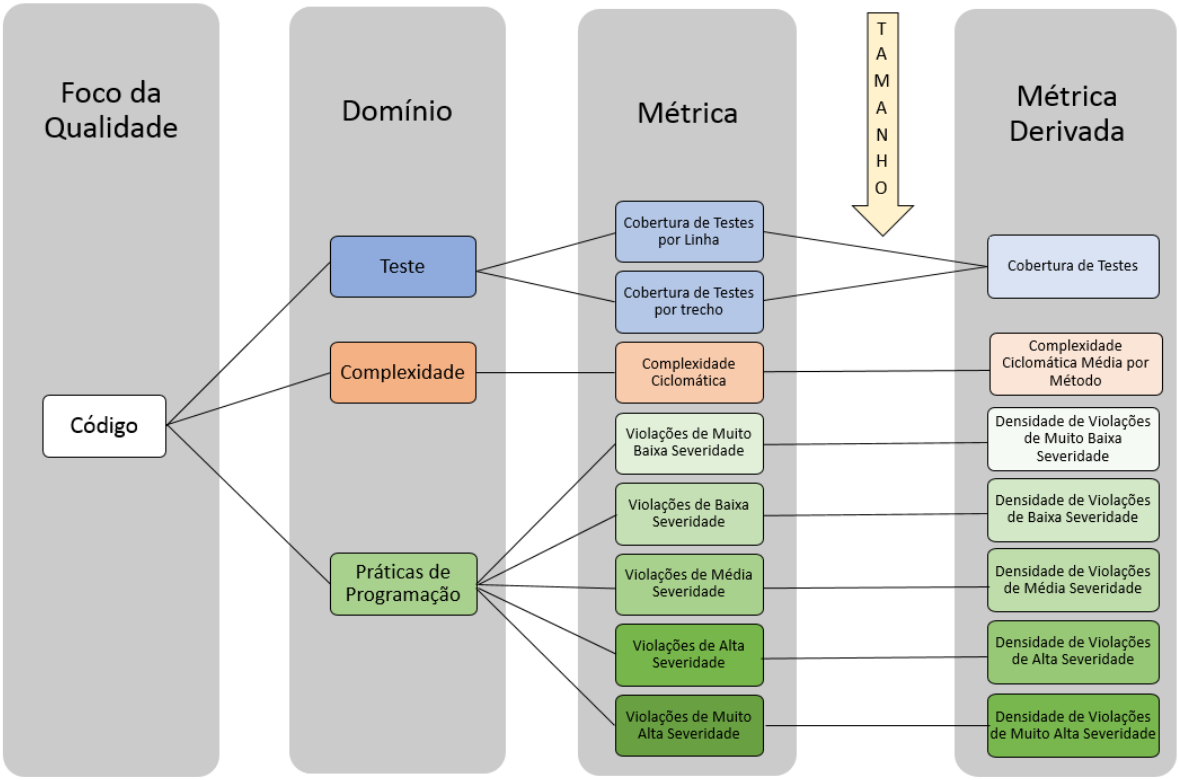


Figura 11- Esquema de Métricas

Tabela 5- Métricas Derivadas

TESTES DE SOFTWARE	
COBERTURA DE TESTES (CT)	
Objeto da Medição	Software entregue ao Órgão X

Foco de Qualidade	Conhecer o quanto do código recebido pelo Órgão X é coberto através de testes unitários.
Fonte de Coleta	Código fonte do Software entregue
Descrição/ Fórmula	$$CT = \frac{(C_{true} + C_{false} + LC)}{2 * B + EL}$$ Onde: CT = caminhos que foram avaliadas como 'true' pelo menos uma vez CF = caminhos que foram avaliadas como 'false' pelo menos uma vez LC = linhas cobertas (linhas a cobrir – linhas já cobertas) B = número total de caminhos EL = número total de linhas executáveis (linhas a cobrir)
Método/Ferramenta de Coleta	Ferramenta de Análise Estática
COMPLEXIDADE	
COMPLEXIDADE CICLOMÁTICA MÉDIA POR MÉTODO (CC)	
Objeto da Medição	Software entregue ao Órgão X
Foco de Qualidade	Conhecer a Complexidade Ciclomática inerente ao software recebido pelo Órgão X.
Fonte de Coleta	Código fonte do Software entregue
Descrição/ Fórmula	Trata-se, simplificadamente, da contabilização dos ramos (caminhos) possíveis para um trecho de código, contabilizados neste contexto a partir do somatório de palavras chaves referentes a estruturas condicionais e quantidade de laços de repetição passíveis de execução. $$CC = \frac{\sum(CP)}{n}$$ Onde: CP = Todas as palavras reservadas da linguagem indiquem definição de funções, estruturas condicionais e estruturas de repetições. n = Número total de métodos existentes no projeto.
Método/Ferramenta de Coleta	Ferramenta de Análise Estática
PRÁTICAS DE PROGRAMAÇÃO	
VIOLAÇÃO DE SEVERIDADE “s” (V<s>)	

"**s**" pode ser substituído por : S = severidade Muito Baixa B = severidade Baixa M = severidade Média A = severidade Alta X = severidade Muito Alta

Objeto da Medição	Software entregue ao Órgão X
Foco de Qualidade	Conhecer quantas violações de severidade baixa foram encontradas no código por linha de código.
Fonte de Coleta	Código fonte do Software entregue
Descrição/ Fórmula	É a densidade de violações de severidade baixa por linha de código. $$Vs = \frac{QVs}{Locs}$$ Onde: QVs = Quantidade de violações identificadas de severidade <s>. Locs = Linhas de código.
Método/Ferramenta de Coleta	Ferramenta de Análise Estática

3.2.1.2. Proposta de Indicador

As métricas selecionadas para coleta devem complementar os critérios já existentes de verificação de qualidade, sobre os quais atuam multa e glosa dos contratos. A sugestão é que seja utilizado o conceito de "Defeitos de Código" na composição de um "Indicador de Defeitos" que também relacione defeitos funcionais do sistema.

A noção de "Defeito de Código" foi elaborada através do estabelecimento de valores de referência em relação as métricas estáticas a serem coletadas.

Os valores de referência foram compostos a partir de literatura relacionada e convenções utilizadas no mercado. Para complexidade Ciclomática, assim como para Cobertura de Teste não há um valor absoluto de referência, porém o valor 5 para Complexidade Ciclomática é bem aceito como valor limite (JQANA, 2015), assim como cobertura de Código com valor mínimo de 80% (HUMBLE; FARLEY, 2014).

Os pesos atribuídos foram estabelecidos a partir do valor limite de defeitos tolerados pelo Órgão X: 0,05, que passarão a valer em breve.

Tabela 6- Valores de Referência

CC- Complexidade Ciclomática Média por Método		
Valores de Referência	**Status**	**Peso**
0≤ CC <5	Ótimo	0
5≤ CC <10	Bom	0,01
10≤ CC <20	Ruim	0,03
20≤ CC	Péssimo	0,05
CT- Cobertura de Testes		
Valores de Referência	**Status**	**Peso**
80 ≤ CT	Ótimo	0
60 ≤ CT <80	Bom	0,01
30 ≤ CT <60	Ruim	0,03
CT <30	Péssimo	0,05
VB- Densidade de Violação de Baixa Severidade		
Valores de Referência	**Status**	**Peso**
0,01 ≥ VB	Ótimo	0
0,01 < VB ≤ 0,05	Bom	0,01
0,05 < VB ≤ 0,1	Ruim	0,03
0,1 < VB	Péssimo	0,05
VM- Densidade de Violação de Média Severidade		
Valores de Referência	**Status**	**Peso**
0,005 ≥ VM	Ótimo	0
0,005 < VM ≤0,025	Bom	0,01
0,025 < VM ≤ 0,05	Ruim	0,03
0,05 < VM	Péssimo	0,05
VA- Densidade de Violação de Alta Severidade		
Valores de Referência	**Status**	**Peso**
VA = 0	Ótimo	0
VA ≤ 0,002	Bom	0,01
0,002 < VA ≤ 0,005	Ruim	0,03
0,005 < VA	Péssimo	0,05
VX- Densidade de Violação de Muito Alta Severidade		
Valores de Referência	**Status**	**Peso**
VX = 0	Ótimo	0
VX>0	Péssimo	0,05

3.2.2.Ferramentas

As duas ferramentas que compõe esta estratégia de verificação de qualidade são o ***Jenkins*** e o ***SonarQube*** (Figura 12).

Jenkins é uma ferramenta de integração contínua focada em construção de build e execução de testes continuamente ao longo do projeto, garantindo a integração

de mudanças diárias (JENKINS, 2015). *SonarQube* é uma ferramenta aberta para gerenciamento de qualidade de código, através de métricas, que consegue analisar mais de 20 linguagens diferentes (SONAR, 2014).

Figura 12- Símbolo das ferramentas adotadas. (SONAR; JENKINKS, 2015)

Neste trabalho o *SonarQube* foi escolhido dentre outras ferramentas semelhantes por ser uma ferramenta com grande difusão no mercado, além de ser a plataforma onde o Perfil de Qualidade para Órgãos da APF foi customizado (CROZARA, 2014).

A escolha do *Jenkins* se deu a diversos fatores, alguns deles foram (JENKINS, 2015):

- Fácil Instalação;
- Fácil Configuração;
- Lista de mudanças entre builds;
- Execução de testes;
- Suporte a diversos plugins (+400);
- Envio de e-mail e/ou SMS caso build falhe;

Jenkins é uma ferramenta de fácil configuração, onde, independente da linguagem, é possível adicionar e retirar quaisquer passos pré-build, pós-build ou durante os builds. Entre estes passos, é possível fazer as execuções dos testes de dentro do próprio *Jenkins* e rodar as análises do *SonarQube* e enviar para o *Dashboard* do próprio Sonar. Estas são características muito importantes para a estratégia aqui definida.

Outro grande motivo para a escolha do *Jenkins* foi à possibilidade de tornar a verificação de qualidade de código a mais automatizada possível, pois ele oferece a possibilidade de gerar um build automaticamente a cada *commit*, caso desejado. Isso permite que o *SonarQube* seja executado no momento em que o código vai para o repositório e já disponibilize suas análises sobre aquele código, possibilitando assim uma atualização contínua das métricas em tempo de execução de projeto.

3.2.3.Papéis

Os papéis escolhidos para desempenhar os procedimentos da estratégia foram selecionados conforme os já estabelecidos no processo GeDDAS (Figura 7), com objetivo de aproveitar as capacidades já disponíveis no Órgão X, são eles:

I. **Equipe de Qualidade** – Responsável por apoiar a garantia da qualidade e emitir parecer técnico dos produtos ou serviços produzidos no processo;

   Atribuições na Estratégia: Dentro da estratégia de verificação, emitiria um relatório de acordo com a análise de qualidade de código feita pelo *SonarQube* e enviaria este para o Gerente de Projeto. Para a estratégia, precisam ter conhecimentos de como gerar os indicadores propostos aqui a partir da análise do *SonarQube*.

II. **Time de Desenvolvimento:** Composto por uma média de 3 a 8 profissionais multifuncionais que produzem um incremento de software potencialmente utilizável (“Pronto”) ao final de cada *Sprint*. Deve ser uma equipe auto-organizada;

   Atribuições na Estratégia: A princípio esta equipe não precisa ter conhecimentos específicos sobre qualidade de código nem como ela é calculada no *SonarQube*, mas é necessário existir noções de qualidade para serem aptos a interpretar os resultados obtidos.

III. **Gestor de Projetos do Órgão**: É um papel dentro do próprio Órgão X, responsável por fornecer informações relacionadas à área de TI; auxiliar no planejamento; identificar riscos, restrições e premissas sobre o ambiente da área de TI.

   Atribuições na Estratégia: Responsáveis por receber o relatório e determinar quais medidas serão tomadas mediante o resultado do relatório e as especificações de qualidade do projeto pré-determinadas em contrato. Caso necessário, aplicaria multas a empresa de desenvolvimento contratada.

3.2.4.Procedimentos

A composição desta estratégia foi constituída de forma a impactar o menos possível no desenvolvimento, sendo a mais automatizada possível para o Órgão X e para a Equipe de Qualidade, e mantendo a transparência dos resultados gerados pela ferramenta para todos os interessados no painel do *SonarQube*.

Para a execução destes procedimentos, como será apresentado abaixo, alguns pré-requisitos precisam ser cumpridos, são eles:

- Projeto configurado conforme necessidade do *SonarQube*;
- Procedimento (*Job*) no *Jenkins* configurado para monitorar repositório;
- *Jenkins* com credenciais de acesso ao repositório;
- Equipe enviando incrementos de software ao repositório especificado;
- Todos os papéis possuindo acesso ao *SonarQube* e ao *Jenkins*;

Abaixo está um diagrama, exemplificando os procedimentos que acontecem durante projeto que fazem com que uma análise ocorra, e quais passos são executados por alguns papéis a partir do momento que a análise está pronta.

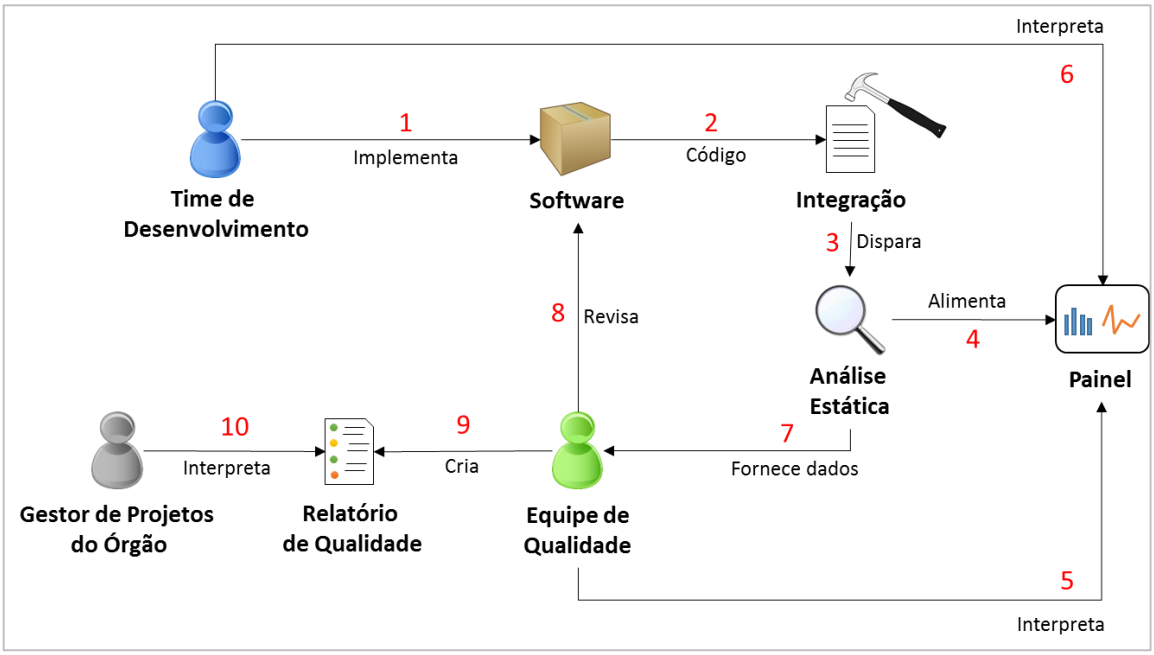


Figura 13- Estratégia de Verificação.

Estes estão os procedimentos que envolvem a estratégia. Resumidamente, durante o desenvolvimento do projeto, à medida que os desenvolvedores implementam o software, gerando código e adicionando ao repositório, o *Jenkins*, ferramenta de integração, detectando novos incrementos de software, dará início a sua construção de build, gerando o executável do arquivo, logo depois irá também executar os testes do projeto e por fim invocará o *SonarQube* para realização da análise estática. O Sonar, por sua vez, irá disponibilizar os resultados da sua análise no painel, que está disponível para consulta e interpretação tanto dos próprios desenvolvedores (Time de Desenvolvimento) quanto da Equipe de Qualidade e dos Gestores do órgão. A equipe de qualidade irá criar um relatório com base nos dados fornecidos pelo *SonarQube* e enviá-lo ao Gestor de Projeto, que irá utilizar este relatório para tomar as decisões necessárias em relação a qualidade de código.

Os passos executados na Figura 13, são detalhados da seguinte forma:

1. Os incrementos de software desenvolvidos pelo Time de Desenvolvimento são disponibilizados no repositório;
2. Código novo é detectado pela ferramenta de integração contínua (*Jenkins*);
3. Ferramenta de integração contínua dispara a execução da análise estática (*SonarQube*);
4. O painel do projeto é alimentado com os resultados;
5. Equipe de Qualidade tem acesso aos resultados do painel;
6. Time de Desenvolvimento tem acesso aos resultados do painel;
7. *SonarQube* fornece dados para Equipe de Qualidade;
8. Equipe de Qualidade revisa software conforme demais critérios estabelecidos;
9. Equipe de Qualidade cria um Relatório de Qualidade de Código baseada nos dados obtidos através da ferramenta de análise estática;
10. Relatório é disponibilizado para o Gestor de Projetos, para tomadas de decisões necessárias em relação a qualidade de código no projeto;

Os procedimentos de criação de relatório (9) e disponibilização para o Gestor de Projetos (10) podem ser executados na periodicidade em que o Órgão necessitar. No contexto do processo GeDDAS, sugere-se que ocorra a nível de *Release*. Os demais procedimentos são contínuos, ou seja, a cada atualização disponibilizada no repositório estes podem ser realizados.

3.3. VALIDAÇÃO DA ESTRATÉGIA

A validação da estratégia se dará de duas formas: funcionalidade da estratégia (Validação Funcional) e validação da percepção perante os envolvidos no Órgão X (Validação Qualitativa).

3.3.1. Validação Funcional

A validação funcional será, exatamente, verificar se as execuções necessárias das ferramentas foram realizadas da forma esperada. Isso quer dizer, para a ferramenta de integração será observado:

1) Se a ferramenta de integração detectou atualização do repositório;
2) Se a ferramenta de integração iniciou a construção;
3) Se a ferramenta de integração gerou o executável;
4) Se a ferramenta de integração executou os testes; e
5) Se a ferramenta de integração invocou a ferramenta de análise estática.

Por parte da ferramenta de análise estática, será observado:

6) Se a ferramenta de análise estática realizou a inspeção;
7) Se a ferramenta de análise estática conseguiu encontrar o relatório de testes gerado na execução dos testes pela ferramenta de integração contínua; e
8) Se a ferramenta de análise estática enviou os resultados para o painel correto do projeto no servidor;

Além das ferramentas, os indicadores serão calculados para analisar seus efeitos sobre o projeto piloto.

3.3.2. Validação Qualitativa

Para validação dentro do Órgão X, serão realizadas entrevistas semiestruturadas (Apêndice 1) com alguns dos papéis chave da estratégia, Gestor de Projetos do Órgão X e Equipe de Qualidade, estes usufruíram em maior proporção dos resultados da estratégia. Estas entrevistas tentarão identificar:

1) Nível de aceitação da estratégia;
2) Se está satisfatório a forma com a qual ela ocorre;
3) Se as ferramentas (principalmente o *SonarQube*), conseguem ser acessadas e os dados facilmente identificados;
4) Se os passos de execução da estratégia foram claros; e
5) Se está atingindo os objetivos esperados pelo Órgão X;

3.4. RESULTADOS OBTIDOS

Com a estratégia definida, ocorreu sua aplicação em um projeto-piloto dentro do Órgão X, para realizar sua validação. Ela teve apenas uma iteração até o presente momento e, de acordo com a validação da estratégia proposta, os resultados para a ferramenta de integração adotada foram:

1) Observar se a ferramenta de integração detectou atualização do repositório (Figura 14).

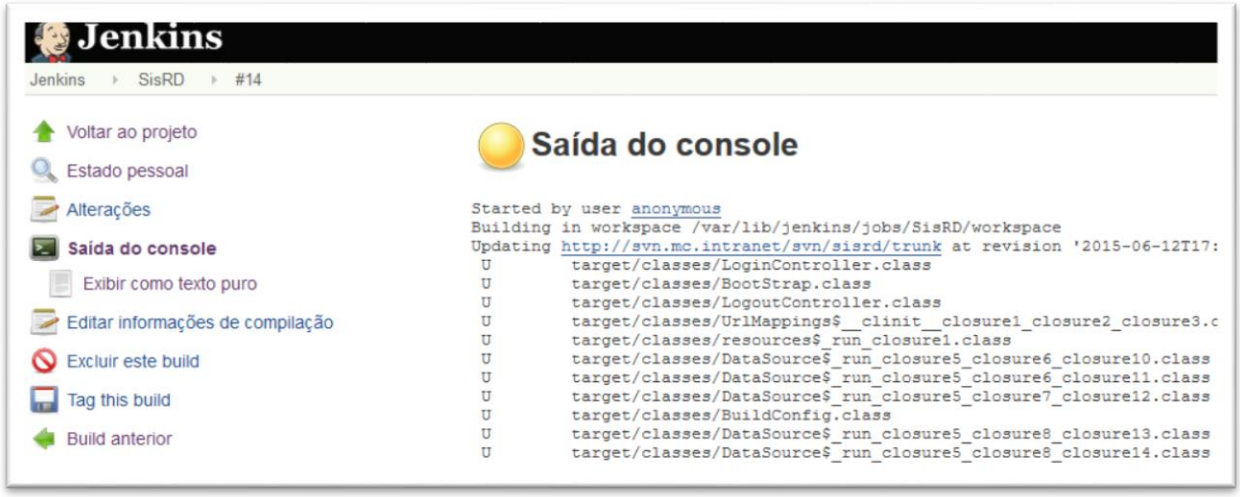


Figura 14- Tela de saída do Console do *Jenkins*.

2) Observar se a ferramenta de integração iniciou a construção. (Figura 15)

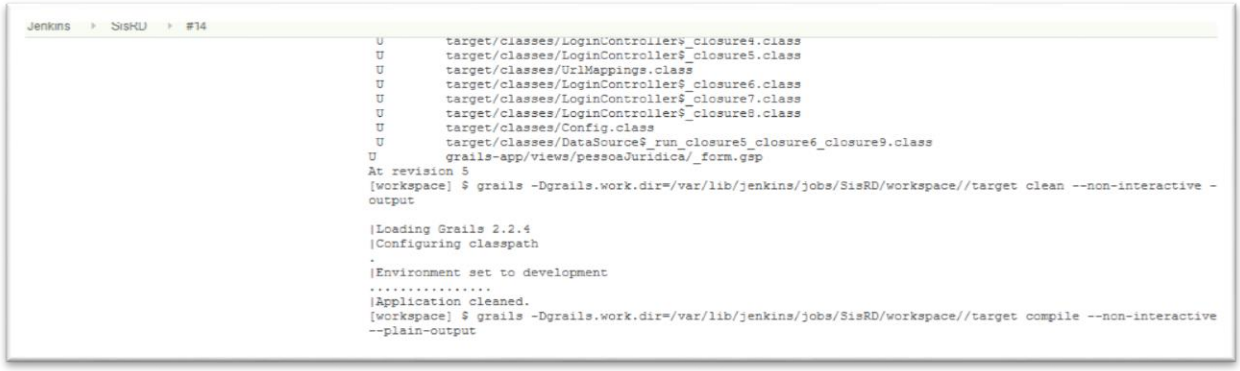


Figura 15- Saída do Console do *Jenkins* iniciando Construção.

3) Observar se a ferramenta de integração gerou o executável (Figura 16).

```
|Loading Grails 2.2.4
|Configuring classpath
.
|Environment set to production
..............................
|Packaging Grails application
..
|Compiling 223 source files

..
|Compiling 43 source files

................
|Compiling 47 GSP files for package [sisrd]
..
|Compiling 12 GSP files for package [ajaxdependancyselection]
.
|Compiling 4 GSP files for package [databaseMigration]
.
|Compiling 4 GSP files for package [mail]
.
|Compiling 1 GSP file for package [jasper]
.
|Compiling 2 GSP files for package [auditLogging]
..
|Building WAR file
.....................................................................................................
|Done creating WAR target/sisrd-0.1.war
```

Figura 16- Saída do Console do *Jenkins* gerando Build.

4) Observar se a ferramenta de integração executou os testes (Figura 17).

```
|Completed 43 unit tests, 35 failed in 12261ms
.........................
|Packaging Grails application
.................................Tests FAILED
|

............
|Cobertura Code Coverage Complete (view reports in: /var/lib/jenkins/jobs/SisRD/workspace/target/test-reports
/cobertura)
```

Figura 17- Saída do Console do *Jenkins* finalizando os testes.

5) Observar se a ferramenta de integração invocou a ferramenta de análise estática (Figura 18).

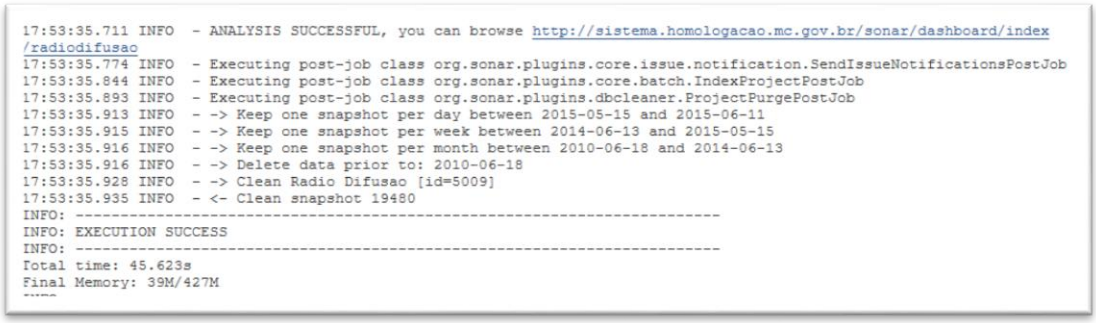


Figura 18- Saída do Console do *Jenkins* executando o *SonarQube*.

6) Observar se a ferramenta de análise estática realizou a inspeção (Figura 19).
7) Observar se a ferramenta de análise estática conseguiu encontrar o relatório de testes gerado na execução dos testes pela ferramenta de integração contínua (Figura 19).
8) Se a ferramenta de análise estática enviou os resultados para o painel correto do projeto no servidor (Figura 19).

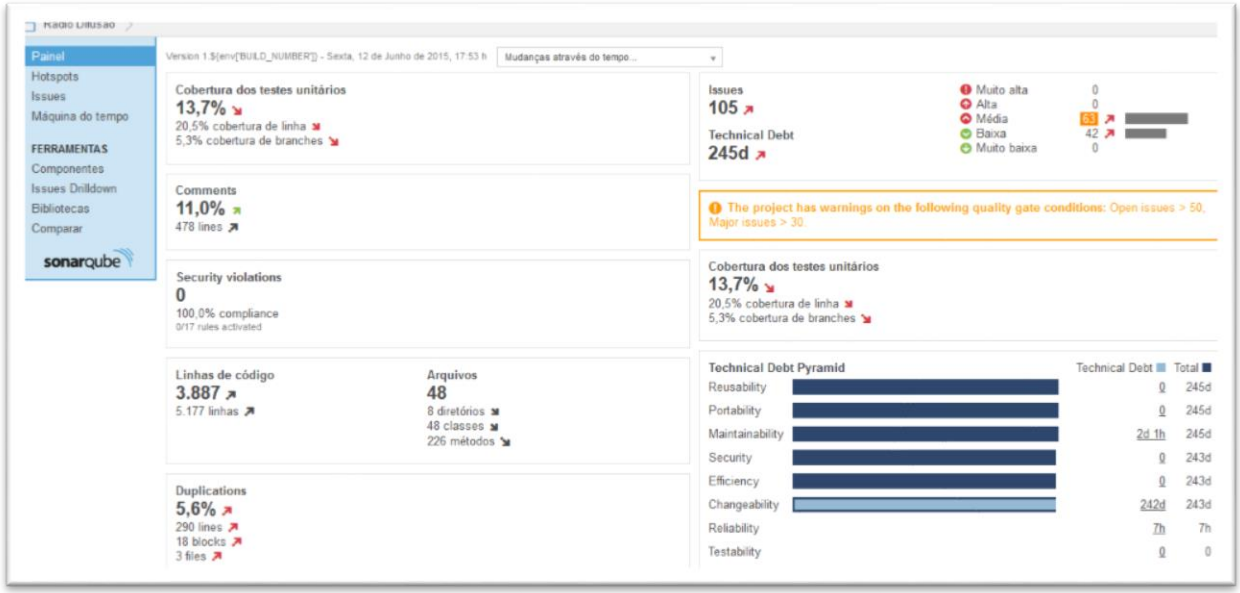


Figura 19- Resultado da Análise no *SonarQube*.

É possível observar no painel de um projeto no Sonar, o nome do projeto, sua cobertura de testes, linhas de código presentes, porcentagem de comentários, e as violações que foram detectadas naquela análise. Mais abaixo é apresentado também a complexidade do projeto por método, classe e arquivo.

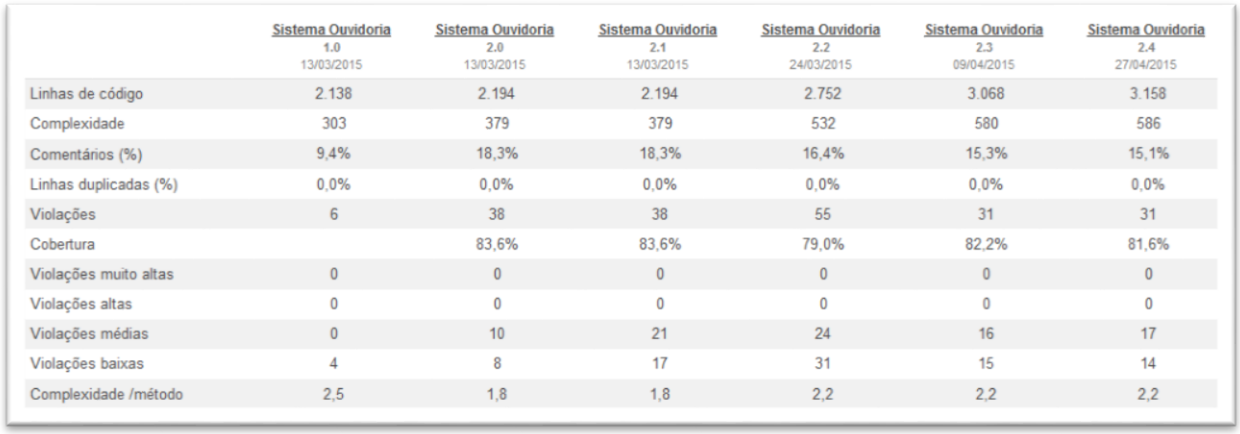
No projeto-piloto atual, onde está sendo testada de fato a estratégia de verificação, apenas uma *sprint* ocorreu. Em seguida pode ser visto o resultado do Sonar para a Sprint 1, na Figura 20, e logo em sequência o resultado dos indicadores na Tabela 7.

Figura 20- Resultado da Análise do Projeto-Piloto Atual.

Tabela 7- Fator de Defeitos da Sprint 1 do Projeto-Piloto

Fator de Defeitos Projeto							
	CC	**CT**	**VB**	**VM**	**VA**	**VX**	**Total Sprint**
Sprint 1	0	0.05	0.01	0.01	0	0	0.07

A coleta de dados dos indicadores também foi feita no projeto-piloto anterior. Este foi acompanhado do começo ao fim, assim é possível calcular todos os indicadores para todas as *sprints* do projeto e mostrar a evolução do mesmo e como os indicadores se comportam atualmente, já que no piloto atual eles serão refinados. Abaixo há todos os dados coletados do projeto na Figura 21, e logo em seguida os resultados dos indicadores para cada Sprint e o total da Sprint na Tabela 8.



	Sistema Ouvidoria 1.0 13/03/2015	Sistema Ouvidoria 2.0 13/03/2015	Sistema Ouvidoria 2.1 13/03/2015	Sistema Ouvidoria 2.2 24/03/2015	Sistema Ouvidoria 2.3 09/04/2015	Sistema Ouvidoria 2.4 27/04/2015
Linhas de código	2.138	2.194	2.194	2.752	3.068	3.158
Complexidade	303	379	379	532	580	586
Comentários (%)	9,4%	18,3%	18,3%	16,4%	15,3%	15,1%
Linhas duplicadas (%)	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
Violações	6	38	38	55	31	31
Cobertura		83,6%	83,6%	79,0%	82,2%	81,6%
Violações muito altas	0	0	0	0	0	0
Violações altas	0	0	0	0	0	0
Violações médias	0	10	21	24	16	17
Violações baixas	4	8	17	31	15	14
Complexidade /método	2,5	1,8	1,8	2,2	2,2	2,2

Figura 21- Todas as Análises e resultados do SisOuvidoria.

Tabela 8- Resultado Fator de Defeitos por Sprint do Projeto-Piloto Anterior.

Fator de Defeitos Projeto							
	CC	**CT**	**VB**	**VM**	**VA**	**VX**	**Total Sprint**
Sprint 1	0	0	0	0	0	0	0
Sprint 2	0	0	0	0.01	0	0	0.01
Sprint 3	0	0.01	0.01	0.01	0	0	0.03
Sprint 4	0	0	0	0.01	0	0	0.01
Sprint 5	0	0	0	0.01	0	0	0.01

Sobre a percepção dos papéis Gestor de Projetos do Órgão X e Equipe de Qualidade, ambos concordam que objetivos da estratégia estão claros e alinhados as novas perspectivas do órgão, e contribui positivamente para a atual conjuntura. Algumas afirmações que se destacaram foram:

“A estratégia muda a visão do órgão em relação a qualidade de software completamente, hoje vivemos um verdadeiro caos, esta estratégia permite um *up* acima das expectativas. ” (Gestor de Projetos)

“Acredito que os objetivos estão claros, e que esta estratégia vai de encontro direto com uma das grandes necessidades atuais aqui dentro. ” (Gestor de Projetos)

“A estratégia está muito boa, mas falta uma relação direta com termos contratuais e procedimentos internos. ” (Equipe de Qualidade)

“Hoje, com base nos contratos, nós só olhamos para papel, acredito que olhar de verdade para o software é mais do que bom, é necessário. ” (Equipe de Qualidade)

“O mais legal é que não causa grande impacto, pois a maior parte é automática, assim temos adição de valor a um baixíssimo custo e isso é louvável, pois essa informação vai ajudar a melhorar a qualidade dos códigos aqui dentro e consequentemente a manutenibilidade. ” (Equipe de Qualidade)

4. CONSIDERAÇÕES FINAIS

Neste trabalho foi apresentada uma estratégia de verificação de qualidade de código para a Administração Pública Federal, que permite maior domínio e visibilidade do órgão sobre os serviços e produtos adquiridos.

As principais ações para o desfecho deste trabalho foram uma revisão de literatura sobre os assuntos correlatos, e um estudo de caso utilizando como objeto de estudo um órgão da APF.

A revisão de literatura possibilitou a contextualização a respeito de contratações na APF e processos de software, além de consolidação de conhecimentos sobre qualidade de software, verificação de software e integração contínua.

No estudo de caso foi realizado um diagnóstico do contexto do Órgão estudado, e partir de então foi proposta a Estratégia de Verificação de Qualidade de Código, principal produto deste trabalho, e que é composta por métricas, papéis, ferramentas e procedimentos.

O diagnóstico do órgão evidenciou as principais lacunas nos processos do Órgão X no que diz respeito a qualidade, principalmente de código, porém as métricas e valores de referência propostos já representam um significante instrumento para avaliação dos produtos adquiridos, podendo ser utilizadas durante a fase de Gestão de Contratos.

A infraestrutura disponibilizada torna possível a execução da estratégia e garante a agilidade das verificações e monitoramento constante de código. Além disso, as ferramentas selecionadas são extensíveis e contam com uma comunidade bastante ativa, o que favorece melhorias constantes no ambiente.

Os papéis selecionados já são ativos do Órgão e praticam atribuições compatíveis com as previstas pela estratégia, o que reforça a viabilidade de execução da estratégia.

Os procedimentos adotados pela estratégia são leves para os papéis envolvidos, pois os cálculos complexos e volumosos são automatizados pelas ferramentas de integração e análise estática.

Desta forma a estratégia mostrou-se viável, porém trata-se de uma solução inicial, que ainda deve ser refinada.

4.1. TRABALHOS FUTUROS

Há algumas propostas de trabalhos futuros com o que foi construído até o presente momento, a primeira delas é utilizar o *Jenkins* mais efetivamente, por ser uma ferramenta de integração contínua, realizando o *deploy* da aplicação no servidor do Órgão X, e preparando-o para testes funcionais. Esta ação já está sendo viabilizada, em conjunto com uma ação de melhoria de gestão de configuração de software.

Até o fim do projeto-piloto, os *Indicadores* serão mais refinados, adicionando novas métricas e, principalmente, automatizando-os juntamente ao *SonarQube* (ferramenta de análise estática), já que os cálculos de densidade de violações foram realizados manualmente, apesar de a ferramenta fornecer todos os dados para seu cálculo.

Alguns ajustes em nível de ferramenta ainda estão sendo realizados para calibrar quais são os melhores momentos de atualização das versões das análises para promover uma visualização mais significativa no painel da ferramenta de análise estática.

REFERÊNCIAS BIBLIOGRÁFICAS

ALBRECHT, A. J. GAFFNEY, J. E. **Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation**. IEEE Transactions on Software Engineering, 1983.

AL-QUTAISH, R. E. **Quality models in software engineering literature: an analytical and comparative study**. Journal of American Science, v. 6, n. 3, p. 166–175, 2010.

BECK, K. **Implementation Patterns**. Addison Wesley, 2007.

BENKLER, Y. **The Wealth of Networks: How Social Production Transforms Markets and Freedom**. Yale University Press, 2006.

BERGEY, J. K.; CLIMENTS, P. C. **Software Architecture in DoD Acquisition: An Approach and Language for a Software Development Plan**. Software Engineering Institute, fev. 2005.

BIEMAN, J. M. OTT, L. M. **Measuring Functional Cohesion**. IEEE Transactions on Software Engineering, 1994.

BLANCHARD, B. **Logistics engineering and management**. 4. ed. [s.l.] Englewwod Cliffs: Prentice Hall, 1992.

BOEHM, B. W., BASILI, V.R., 2001, "**Software Defect Reduction Top 10 List**.", IEEE Computer 34 (1): 135-137.

BOEHM, B. W., BROWN, J. R., KASPAR, H., LIPOW, M., MCLEOD, G., MERRITT, M. **Characteristics of Software Quality**. North Holland Publishing, Amsterdam, The Netherlands, 1978.

BOEHM, B. **Software Engineering Economics**. Prentice-Hall, 1981.

BRASIL. Acórdão No 2314/2013. 2013. **Levantamento de Auditoria.** Tribunal de Contas da União. 2013. Disponível em: <https://contas.tcu.gov.br>

_____. **Contrato n 05/2012-MC,** Ministério das Comunicações. 2012a. Disponível em: <http://www.comunicacoes.gov.br/licitacoes-e-contratos/editais-e-avisos/cat_view/17-editais-e-avisos/111-contratos-vigentes/117-2012?start=20>

_____. **Decreto-Lei Nº 200, de 25 de fevereiro de 1967**. 1967. Disponível em: <http://www.planalto.gov.br/ccivil_03/decreto-lei/del0200.htm>. Acesso em: 20 ago. 2014

_____. **Decreto Nº 2.271, de julho de 1997**. 1997. Disponível em: <http://www.planalto.gov.br/ccivil_03/decreto/d2271.htm>. Acesso em: 3 set. 2014.

_____. **Guia de boas práticas em contratação de soluções de tecnologia da informação: riscos e controles para o planejamento da contratação**. Tribunal de Contas da União, 2012b.

_____. **Guia Prático para Contratação de Soluções de Tecnologia da Informação**. Ministério do Planejamento Orçamento e Gestão. Secretaria de Logística e tecnologia da informação (SLTI). 2011. V. 1.1. 2011.

_____. **Instrução Normativa No. 04. 2014**. Disponível em: <http://www.governoeletronico.gov.br/biblioteca/arquivos/in-nb0-4-2014/view >. Acesso em: 20 ago. 2014a.

_____. **Lei no8.666, de 21 de junho de 1993**. 1993. Disponível em: <http://www.planalto.gov.br/ccivil_03/leis/l8666cons.htm>. Acesso em: 22 ago. 2014

_____. **Lei 10.520, de 17 de julho de 2002**. 2002. Disponível em: <http://www.planalto.gov.br/ccivil_03/leis/2002/l10520.htm>. Acesso em: 3 set. 2014.

_____**Plano Estratégico de Tecnologia da Informação (PETI)** , Ministério das Comunicações.2014. Disponível em: <http://www.mc.gov.br/index.php?option=com_mtree&task=att_download&link_id=680&cf_id=24 >. Acesso em: 10 out. 2014b

_____. SISP — **Programa de Governo Eletrônico Brasileiro -Sítio Oficial**. 2014. Disponível em: <http://www.governoeletronico.gov.br/sisp-conteudo>. Acesso em: 23 set. 2014c.

_____. SLTI — **Programa de Governo Eletrônico Brasileiro -Sítio Oficial**. 2014d. Disponível em: <http://www.governoeletronico.gov.br/o-gov.br/secretaria-de-logistica-e-tecnologia-da-informacao>. Acesso em: 23 set. 2014d.

_____. **Termo de Referência - Anexo I, Edital de Pregão Eletrônico n 038/2011-MC**, Ministério das Comunicações. 2011b. Disponível em: <http://comunicacoes.gov.br/o-ministerio/172-editais-e-avisos/23912-edital-de-pregao-eletronico-no-0382011-mc>

_____**Termo de Referência - Anexo V Processo de Gestão de Demandas de Desenvolvimento Ágil de Software do Ministério das Comunicações**. Ministério das Comunicações. 2014e.

CAMPBELL, D. T.; STANLEY, J. C. **Delineamentos experimentais e quase experimentais de pesquisa**. Editora da Universidade de São Paulo, SP, 1979.

CHEMUTURI, M. **Mastering Software Quality Assurance**. 2010

COHEN, D., LINDVALL, M., COSTA, P. (2004). **An introduction to agile methods**. InAdvances in Computers (pp. 1-66). New York: Elsevier Science.

CROZARA, K. H. **Monitoração da qualidade de produto nas contratações de soluções de TI da Administração Pública Federal**. [S.l.], 23 ago. 2014. Monografia (Bacharelado em Engenharia de software) —Universidade de Brasília, Brasília, 2014

CRUZ, C. S. DA; ANDRADE, E. L. P. DE; FIGUEIREDO, R. M. DA C. **Processo de Contratação de Serviços de Tecnologia da Informação para Organizações Públicas**. Ministério da Ciência e Tecnologia. Secretaria de Política de Informática, 2011.

DE BIAZZI, M. R.; MUSCAT, A. R. N.; DE BIAZZI, J. L. **Process management in the public sector: A Brazilian case study. Management of Engineering & Technology, 2009. PICMET 2009**. Portland International Conference on. Anais...IEEE, 2009. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5261781>. Acesso em: 27 set. 2014.

DEISSENBOECK, F. et al. **Tool Support for Continuous Quality Control**. **IEEE Software**, v. 25, n. 5, p. 60–67, set. 2008.

DERN, E. **Out of the Crisis**, Cambridge, London, 1982.

EISENHARDT, K. M. **Building Theories from Case Study Research**. Academy of Management Review, v. 14, p. 532–550, 1989.

FEIGENBAUM, A. V. "**Total Quality Control**", McGraw-Hill, 1983.

FELIZARDO, K. R. **Apoio computacional para inspeção de software**. Revista de Ciência da Computação, Lavras, MG, v. 3, 2004.

FENTON, N., Pfleeger, S.L.: **Software Metrics (2nd ed.), a rigorous and practical approach**, PWS Publishing Co., Boston, MA, USA (1997)

_____; MELTON, A. **Deriving Structurally Based Software Measures**. Jornal of System and Software, 1990).

FIELDS. J. **The Maintainability of Unit Tests**. Disponível em: <http://blog.jayfields.com/2010/02/maintainability-of-unit-tests.html>

FOWLER, M. **Continuous Delivery**. Disponível em: <http://martinfowler.com/bliki/ContinuousDelivery.html>. Acesso em: 1 mai. 2015.

_____. **Continuous Integration**. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 1 mai. 2015.

FUGETTA, A. **Software Process: A Roadmap. The Future of Software Engineering**, 2000.

GERHARDT, T.; SILVEIRA, D. T. **Métodos de pesquisa-coordenado pela Universidade Aberta do Brasil – UAB/UFRGS e pelo Curso de Graduação Tecnológica – Planejamento e Gestão para o Desenvolvimento Rural da SEAD/UFRGS** Porto Alegre: Editora da UFRGS, 2009

GIL, A. C. **Como elaborar projetos de pesquisa**. São Paulo: Atlas, 2008.

GOUSIOS, G. KARAKOIDAS, V. STROGGYLOS, K. LOURIDAS, P. VLACHOS, V. **Software Quality Assesment of Open Source Software**. 2007.

HALSTEAD, M. H. **Natural Laws Controlling Algoritmic Structures?**, 1972.

HASHIURA, H.; MATSUURA, S.; KOMIYA, S. **A tool for diagnosing the quality of Java program and a method for its effective utilization in education**. In: PROCEEDINGS OF THE 9TH WSEAS INTERNATIONAL CONFERENCE ON APPLICATIONS OF COMPUTER ENGINEERING. World Scientific and Engineering Academy and Society (WSEAS), 2010.

HAUSEN, L. H. **Yet Anothe Software Quality and Productivity Modeling - YAQUAPMO**. Em Proceedings of the Twenty-Second Annual Hawaii International Conference on System Science, volume 2, páginas 978-987, 1989.

HIGHSMITH, J.**“Agile Software Development Ecosystems**,” Addison–Wesley, Boston, MA, 2002.

HUMBLE, J. FARLEY, D. **Entrega Contínua: Como Entregar Software**. 2014.

HUMPHREY, W. S. **Managing the Software Process**. Addison-Wesley Publishing Company, Inc. 1989.

IEEE. **IEEE Standard for Software Maintenance**. IEEE Std 1219-1993, p. 1–45, jun. 1993.

_____. **IEEE Standard for a Software Quality Metrics Methodology**. IEEE Std 1061-1998, dez. 1998.

ISO/IEC. **Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE**. [s.l: s.n.] 2014.

JENKINS. **Hudson’s future | Jenkins CI**. Disponível em: <http://jenkins-ci.org/content/hudsons-future>. Acesso em: 12 mai. 2015.

_____. **Plugins**. Disponível em: <https://wiki.jenkins-ci.org/display/JENKINS/Plugins#Plugins-Buildreports>. Acesso em: 12 mai. 2015

JIANG, Y., Cuki, B., MENZIES, T., BARTLOW, N.: **Comparing Design and Code Metrics for Software Quality Prediction**, PROMISE'08: Proceedings of the 4th international workshop on Predictor models in software engineering, pp. 11-18, Leipzig, Germany (2008).

JQANA. **JQANA Report: Metrics Limits**. Disponível em: < https://github.com/cleuton/jqana/wiki/How-to-analyze-jQana-report >. Acesso em: 12 mai. 2015

KAFURA, D. HENRY, S. **Software Quality Metrics Based on Interconnectivity**, 1981.

KUMAR. A. **How to Measure Code Maintainability with Sonar**. Disponível em: <http://vitalflux.com/measure-code-maintainability-sonar/>

LOURIDAS, P. **Static code analysis**. IEEE Software, v. 23, n. 4, p. 58–61, jul. 2006.

MCCABE, T. J. **A Complexity Measure**. IEEE Transactions on Software Engineering, v. SE-2, n. 4, p. 308–320, dez. 1976.

MCCONNELL, S.: **Code Complete**, 2nd Edition, Microsoft Press, Redmond, WA (2004)

MCFEELEY, B., "**IDEAL: A User's Guide for Software Process Improvement**," Ft. Belvoir: Defense Technical Information Center 1996.

MEIRELLES, P. R. M. **Monitoramento de Métricas de código-fonte em projetos de Software Livre**, 2013.

MELO, C. DE O.; FERREIRA, G. R. **Adoção de métodos ágeis em uma Instituição Pública de grande porte-um estudo de caso**. In: Workshop Brasileiro de Métodos Ágeis, Porto Alegre. Anais...2010. Disponível em: <http://agilcoop.org.br/files/WBMA_Melo_e_Ferreira.pdf>. Acesso em: 13 set. 2014.

MENDES, F. F. **Melhoria de Processos de Tecnologia da Informação Multi-Modelo.** Goiânia, 2010. 157 p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

MEYER, M. **Continuous Integration and Its Tools**. **IEEE Software**, v. 31, n. 3, p. 14–16, maio 2014.

MILLS, E. E. **Software metrics.** . [S.l.]: DTIC Document, 1988. Disponível em: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA236140>. Acesso em: 20 set. 2014.

MISRA, S.C.: **Modeling Design/Coding Factors That Drive Maintainability of Software Systems, Software Quality Control**, vol. 13, no. 3, pp. 297-320, Kluwer Academic Publishers, Hingham, MA, USA (2005)

MOREIRA, A.: **Integração Contínua**. [s.d.]. Disponível em: <http://siep.ifpe.edu.br/anderson/blog/?page_id=1015>. Acesso em: 12 mai. 2015

NBR ISO/IEC **9126-1**. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. Tecnologia da informação – Qualidade de Produto de Software – Parte 1:Modelo de Qualidade. Rio de Janeiro, 2003.

PAIVA, J. A. DE; SOUZA, F. M. C. DE. **Model contract for outsourcing of maintenance: a principal-agent approach**. Production, dez. 2012. v. 22, n. 4, p. 796–806. . Acesso em: 12 set. 2014.

PAUL, D. **Continuous Integration. [s.l.] Pearson Education India**, 2007.

PIGOSKI, T. M. **Pratical Software Maintenance: Best Practices for Managing your Software Investment.Wiley Computer Publishing**, 1996.

PRADO, E. P. V.; CRISTOFOLI, F. **Resultados da terceirização da tecnologia da informação em organizações brasileiras**. Gestão & Regionalidade, v. 28, n. 84, p. 77-88, 2012.

PRESSMAN, R. S. **Engenharia de Software**. [s.l.] McGraw Hill Brasil, 2011.

ROMBACH, H. D. **Design measurement: some lessons learned**. IEEE Software, v. 7, n. 2, p. 17–25, mar. 1990.

SATO, D., GOLDMAN, A., and KON, F.. **Tracking the evolution of object oriented quality metrics**. In Proceedings of the 8th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2007), pages 84–92, 2007.

SCHWABER, K.; BEEDLE, M. **Agilè Software Development with Scrum**. 2002.

SECCOM, Universidade Federal de Santa Catarina, Dr. Christiane Gresse von Wangenheim, 2009.

SELLERS, B. **Object-Oriented Complexity**. Prentice-Hall, 1996.

SMART, F. J. **Jenkins: The Definitive Guide**. O'Reilly Media, Inc, Sebastopol, 2011.

SMITH, C. P. **A Software Science Analysis of Programming Size**. Em Proceedings of The ACM National Computer Conference. 1980.

SOFTEX. **Guia Geral do MPS.BR - Melhoria de Processo do Software Brasileiro - Guia Geral MPS de Software**, 2012. Disponível em: < http://www.softex.br/wp-content/uploads/2013/07/MPS.BR_Guia_Geral_Software_2012-c-ISBN-1.pdf>. Acesso em: 1 nov. 2014.

SOMMERVILLE, I. **Software Engineering**. 9. ed. [s.l.] Addison-Wesley, 2011.

SONAR. **SonarQube** Disponível em: <http://www.sonarqube.org/>. Acesso em: 18 out. 2014.

SOUZA, L. P. S., FIGUEIREDO, R. M. C., VENSON, E., RIBEIROJR, L. C. M.; Colaboradores KOSLOSKI, R. A. D.; LIMA, T. **A aplicação do framework ágil Scrum em um processo de gestão de projetos de desenvolvimentos de software por terceiros em um órgão público federal brasileiro**. In: 12th CONTECSI International Conference on Information Systems and Technology Management, Sao Paulo. TECSI, 2015.

STORCH, A.; LAUE, R.; GRUHN, V. **Measuring and visualising the quality of models** 2013 IEEE 1st International Workshop on Communicating Business Process and Software Models Quality, Understandability, and Maintainability. set. 2013

THOUGHTWORKS. **Continuous Delivery | ThoughtWorks**. Disponível em: <http://www.thoughtworks.com/pt/continuous-delivery>. Acesso em: 1 jun. 2015.

VANDOREN, E. **C4 Software Technology Reference Guide - A Prototype**, 1997.

YIN, Robert K. - **Case Study Research - Design and Methods**. Sage Publications Inc., USA, 1989

XP, E. P. **Continuous Integration**. Disponível em: <http://www.extremeprogramming.org/rules/integrateoften.html>. Acesso em: 1 jun. 2015.

APÊNDICE

Apêndice I: Roteiro Para Entrevista

Roteiro para Entrevista

Levantamento sobre a percepção da Estratégia de Qualidade

Data: __/__/____
Objetivo: _____

1- Identificação dos Participantes
   Cargo do Entrevistado: _____
   Nome do Entrevistador: _____

2- Procedimentos
   a. Apresentação do pesquisador
   b. Apresentação do objetivo da entrevista
   c. Apresentação de todos os pontos da Estratégia de Verificação de Qualidade
   d. Efetuar questões para discussão

3- Questões
   a. Está claro o objetivo da estratégia de verificação?
   b. Os objetivos estão alinhados com o que o Órgão esperava?
   c. Estão claros os passos envolvidos na estratégia?
   d. Os indicadores estão claros (tanto na sua interpretação quanto na sua forma de cálculo)?
   e. A estratégia, na sua opinião muda a forma com que o Órgão enxerga qualidade de software?
   f. A estratégia, na sua opinião, melhora a forma com que o Órgão faz verificação de qualidade?

ANEXOS

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method	Podem ocorrer falhas quando versões diferentes da linguagem Java são utilizadas.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
"java.lang.Error" should not be extended	Erros relacionados à java.lang.Error e suas subclasses representam situações anormais que são encontradas apenas pela Java Virtual Machine (JVM). O uso de java.lang.Error e suas subclasses pode levar à confusão.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
"object == null" should be used instead of "object.equals(null)"	A variável presente no lado esquerdo da comparação pode ser nula e o uso de object.equals(null) não trata essa situação.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
"switch" statements should have at least 3 cases	Quando não existem três ou mais casos para a instrução switch deve-se substituir por instruções if.	Ineficiência	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
“public static" fields should always be constant	Variáveis que devem ser transformadas em constantes.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Abstract class names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Abstract Class Without Abstract Method	Pode indicar implementação incompleta.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Abstract classes without fields should be converted to interfaces	Convenção adotada a partir da versão 8 da linguagem Java recomenda que classes abstratas sem campos definidos deve ser convertida em uma interface.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Add Empty String	É uma maneira ineficiente de converter qualquer outro tipo ao tipo String. Pode também indicar implementação incorreta ou incompleta.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Anonymous inner classes containing only one method should become lambdas	A partir da versão 8 da linguagem Java recomenda--se a substituição de classes internas anônimas por lambdas.	Ineficiência	Baixa	Modificabilidade	Manutenabilidade
Append Character With Char	Deve-se evitar concatenar caracteres com os métodos que são definidos para serem usados com Strings.	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Array designators "[]" should be on the type, not the variable	Indicadares de arrays devem ser localizados junto ao tipo para uma melhor legibilidade.	Inconsistência de estilo	Muito baixa	Analisabilidade	Manutenabilidade
Assignments should not be made from within sub-expressions	Atribuições em expressões são difíceis de identificar e afetam a legibilidade do código.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Avoid Branching Statement As Last In Loop	Pode indicar um erro de programação e afetam a legibilidade do código.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Avoid Catching Generic Exception	Captura exceções lançadas pela JVM e pode ignorar possíveis problemas.	Bug em potencial	Muito alta	Maturidade	Confiabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Avoid commented-out lines of code	Linhas de código comentada podem confundir e até mesmo se tornar um possível bug ao se remover o comentário por acidente.	Futuro erro de programação	Média	Analisabilidade	Manutenabilidade
Avoid cycle between java packages	Indica alto acoplamento entre pacotes o que afeta a reusabilidade e indica a necessidade de refatoração do código.	Ineficiência	Média	Modificabilidade, Analisabilidade, Testabilidade e Reusabilidade	Manutenabilidade
Avoid Final Local Variable	Nesses casos variáveis finais não são necessárias e devem ser transformadas em variáveis locais.	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Avoid Inline Conditionals	Condicionais em uma mesma linha afetam a legibilidade do código.	Inconsistência de estilo	Muito baixa	Analisabilidade	Manutenabilidade
Avoid instantiating objects in loops	Objetos criados em loops podem ocupar espaço desnecessário na memória.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Avoid Protected Field In Final Class	Campos protegidos em classes finais não podem ser usados em subclasses. Deve-se considerar o uso do modificador privado ou alterar a acessibilidade do pacote.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Avoid Star Import	Deve-se evitar importar pacotes que não serão utilizados.	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Avoid Throwing Null Pointer Exception	Exceções genéricas não dão um diagnóstico preciso do problema e podem ser confundidas com exceções lançadas pela JVM.	Futuro erro de programação	Média	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Avoid too complex class	Classe muito complexa que deve ser refatorada.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Avoid too deep inheritance tree	Árvores heranças muito grandes podem levar a código muito complexo.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Bad comparison of int value with long constant	Compara valores que têm limites diferentes, o que pode levar a erros.	Indicativo de erros de programação	Baixa	Corretude funcional	Adequação funcional
Bad practice - Abstract class defines covariant compareTo() method	Erro ao fazer a sobrecarga do método compareTo().	Bug	Média	Corretude funcional	Adequação funcional
Bad practice - Certain swing methods needs to be invoked in Swing thread	Pode causar problemas na sincronização dos threads.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Check for sign of bitwise operation	Comparação do resultado de operações bitwise com o operador “maior que” pode levar a resultados inesperados	Bug em potencial	Média	Corretude funcional	Adequação funcional
Bad practice - Class defines clone() but doesn't implement Cloneable	Não é necessariamente um erro, mas pode indicar um possível erro de programação	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Bad practice - Class implements Cloneable but does not define or use clone method	Deve-se remover métodos que não são utilizados e reconsiderar a necessidade de implementar a classe Cloneable	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Bad practice - Class inherits equals() and uses Object.hashCode()	Se a classe herda equals() deve-se usar os métodos de hashcode providos por essa classe	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Bad practice - Class is Externalizable but doesn't define a void constructor	Se a classe não define um constructor void a serialização e a deserialização irão falar em tempo de execução	Bug	Média	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Bad practice - Class is not derived from an Exception, even though it is named as such	Pode causar confusão para os usuários da classe.	Inconsistência de estilo	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Class is Serializable but its superclass doesn't define a void constructor	Se a classe não define um constructor void a serialização e a deserialização irão falar em tempo de execução.	Bug	Média	Corretude funcional	Adequação funcional
Bad practice - Class is Serializable, but doesn't define serialVersionUID	Deve-se adicionar explicitamente o serialVersionUID para garantir a interoperabilidade entre versões do código.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Bad practice - Class names shouldn't shadow simple name of implemented interface	Afeta a legilibidade do código e pode causar confusão quando é necessário resolver as referencias das classes.	Inconsistência de estilo	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Bad practice - Class names shouldn't shadow simple name of superclass	Afeta a legilibidade do código e pode causar confusão quando é necessário resolver as referencias das classes.	Inconsistência de estilo	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Bad practice - Clone method may return null	Métodos Clone nunca devem retornar null, o que pode levar a um erro durante a verificação	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Comparator doesn't implement Serializable	Deve-se considerar a necessidade da implementação de Serializable, o que é uma prática de programação defensiva.	Inconsistência de estilo	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Comparison of String objects using == or !=	Faz a comparação da referência ao invés do valor, o que pode levar a erros.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Comparison of String parameter using == or !=	Faz a comparação da referência ao invés do valor, o que pode levar a erros.	Bug em potencial	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Bad practice - Confusing method names	Nomes confusos que podem levar a erros de programação	Futuro erro de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Covariant compareTo() method defined	Erro ao fazer a sobrecarga do método compareTo().	Bug	Média	Corretude funcional	Adequação funcional
Bad practice - Creates an empty jar file entry	Erro na chamada do método closeEntry() que leva a criação de arquivos jar vazios.	Bug	Muito alta	Corretude funcional	Adequação funcional
Bad practice - Creates an empty zip file entry	Erro na chamada do método closeEntry() que leva a criação de arquivos zip vazios.	Bug	Muito alta	Corretude funcional	Adequação funcional
Bad practice - Dubious catching of IllegalMonitorStateException	Possível erro de design.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Bad practice - Empty finalizer should be deleted	Método vazio que deve ser excluído.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Equals checks for noncompatible operand	Uso do método equals com tipos incompatíveis pode levar a comportamento inesperado.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Bad practice - equals method fails for subtypes	Método equals que apresentará erro ao ser usado por subclasses.	Bug	Muito alta	Corretude funcional	Adequação funcional
Bad practice - Equals method should not assume anything about the type of its argument	Apresenta erros caso o método equals não trate comparações com tipos incompatíveis.	Bug	Muito alta	Corretude funcional	Adequação funcional
Bad practice - equals() method does not check for null argument	Podem ocorrer erros durante a comparação se o método equals não checa valores nulos.	Bug em potencial	Muito alta	Corretude funcional	Adequação funcional
Bad practice - Fields of immutable classes should be final	Por convenção, os campos de classes imutáveis também devem ser imutáveis.	Inconsistência de estilo	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Finalizer does not call superclass finalizer	As ações de finalização definidas pela superclasse não serão executadas, o que pode causar comportamentos	Bug em potencial	Muito alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
	inesperados.				
Bad practice - Finalizer does nothing but call superclass finalizer	Método redundante que deve ser removido.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Finalizer nullifies superclass finalizer	Deve-se verificar o uso do finalizador.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Bad practice - Finalizer nulls fields	Erro que anula campos do finalizador.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Bad practice - Iterator next() method can't throw NoSuchElementException	Implementação inadequada que não capacita o método next() a lançar exceções.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method doesn't override method in superclass due to wrong package for parameter	Erro que impede a sobrecarga de parâmetros.	Bug	Alta	Corretude funcional	Adequação funcional
Bad practice - Method ignores exceptional return value	Pode levar a comportamentos inesperados quando o retorno dos métodos não são checados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method ignores results of InputStream.read()	Pode levar a comportamentos inesperados pois ignora valores do input strem.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method ignores results of InputStream.skip()	Pode levar a comportamentos inesperados pois ignora valores do input strem.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method invoked that should be only be invoked inside a doPrivileged block	Quando o código é invocado por códigos que não fazem verificação de permissões deve-se fazer as operações dentro de um bloco doPrivileged para evitar falhas de segurança.	Bug em potencial	Média	Confidencialidade	Segurança

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Bad practice - Method invokes dangerous method runFinalizersOnExit	Chamada de método que pode levar a comportamento inesperado.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method may fail to close database resource	Resulta em ineficiência de performance e pode causar problemas de comunicação com o banco de dados	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Bad practice - Method may fail to close database resource on exception	Resulta em ineficiência de performance e pode causar problemas de comunicação com o banco de dados	Bug	Muito alta	Utilização de recursos	Eficiência de performance
Bad practice - Method may fail to close stream	Resulta em ineficiência de performance e pode causar a perda do manipulador de arquivos	Bug	Muito alta	Utilização de recursos	Eficiência de performance
Bad practice - Method may fail to close stream on exception	Resulta em ineficiência de performance e pode causar a perda do manipulador de arquivos	Bug	Muito alta	Utilização de recursos	Eficiência de performance
Bad practice - Method might drop exception	O método não trata a exceção e também impede que ela seja tratada, o que pode levar a comportamentos inesperados	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method might ignore exception	O método ignora a exceção, o que pode levar a comportamentos inesperados	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Method with Boolean return type returns explicit null	Implementação inadequada que pode levar à NullPointerException	Bug	Alta	Corretude funcional	Adequação funcional
Bad practice - Needless instantiation of class that only supplies static methods	Instanciação desnecessária que deve ser evitada.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Bad practice - Non-serializable class has a serializable inner class	Pode levar a erros em tempo de execução, pois a classe tenta serializar a classe interna.	Bug em potencial	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Bad practice - Non-serializable value stored into instance field of a serializable class	Pode levar a erros em tempo de execução, pois a classe tenta serializar os seus atributos	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Random object created and used only once	Deve-se guardar os valores aleatórios que foram gerados para evitar a repetição de valores	Ineficiência	Média	Corretude funcional	Adequação funcional
Bad practice - serialVersionUID isn't final	Se o UID é usado para serialização, então deve sempre ser do tipo final.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Bad practice - serialVersionUID isn't long	Se o UID é usado para serialização, então deve sempre ser do tipo long	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - serialVersionUID isn't static	Se o UID é usado para serialização, então deve sempre ser do tipo static	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Static initializer creates instance before all static final fields assigned	Cria uma instância da classe antes de todos os campos serem atribuídos, o que pode levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Store of non serializable object into HttpSession	Pode levar a erros quando a sessão é migrada.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Superclass uses subclass during initialization	Possível uso de subclasse que ainda não foi inicializada.	Bug	Média	Corretude funcional	Adequação funcional
Bad practice - Suspicious reference comparison	Compara a referência em vez do valor.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - toString method may return null	Deve-se evitar retornar valores nulos, pois pode levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Bad practice - Transient field that isn't set by deserialization.	Se o campo não é marcado como transient seu valor nunca será atualizado, o que pode levar a comportamentos inesperados	Bug em potencial	Média	Corretude funcional	Adequação funcional
Bad practice - Unchecked type in generic call	Tipos que não são checados podem levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Bad practice - Usage of GetResource may be unsafe if class is extended	Pode levar a resultados inesperados se a classe for herdada em outro pacote.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Bean Members Should Serialize	Boa prática que determina que beans devem ser serializados	Inconsistência de estilo	Muito baixa	Modificabilidade	Manutenabilidade
Big Integer Instantiation	Não é necessário criar instâncias de valores já existentes.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Boolean Get Method Name	Convenção de nomeação que facilita o entendimento do código.	Inconsistência de estilo	Muito baixa	Analisabilidade	Manutenabilidade
Broken Null Check	Checagem que sempre irá lançar NullPointerException.	Bug	Alta	Corretude funcional	Adequação funcional
Call Super In Constructor	Boa prática que determina a chamada do método super() em construtores	Inconsistência de estilo	Muito baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Case insensitive string comparisons should be made without intermediate upper or lower casing	Requer a criação de objetos temporaries, o que não é eficiente.	Ineficiência	Média	Utilização de recursos	Eficiência de Performance
Class variable fields should not have public accessibility	Deve-se respeitar os princípios de encapsulamento.	Inconsistência de estilo	Baixa	Confidencialidade	Seguranca
Classes should not be coupled to too many other classes (Single Responsibility Principle)	Classes com acoplamento alto que devem ser refatoradas.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Collapsible "if" statements should be merged	Condicionais que devem ser mescladas para melhorar a legibilidade do código	Ineficiência	Média	Analisabilidade	Manutenabilidade
Collection.isEmpty() should be used to test for emptiness	Deve-se usar Collection.isEmpty() ao invés de Collection.size() para verificação de coleções.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Comment Size	Linhas de comentários muito longas que devem ser refatoradas.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Comments should not be located at the end of lines of code	Afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Confusing Ternary	Boa prática que recomenda a não utilização de negações em condicionais que possuem else.	Futuro erro de programação	Média	Analisabilidade	Manutenabilidade
Consecutive Appends Should Reuse	Deve-se usar o objeto alvo para melhorar a performance da operação.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Consecutive Literal Appends	Concatenações repetidas que podem afetar a performance, deve-se refatorar.	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Constant names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply	Afeta a legibilidade do código e indicam implementações do tipo "código espaguete"	Futuro erro de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Correctness - "." used for regular expression	Deve-se verificar possíveis erros no uso de expressões regulares.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - A collection is added to itself	Se o hashcode for computado irá resultar em StackOverflowException	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - A known null value is checked to see if it is an instance of a type	O teste irá sempre retornar falso, deve-se verificar se não é um erro de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - A parameter is dead upon entry to a method but overwritten	Atribuição de valores a parâmetros indica erros de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - An apparent infinite loop	Loop infinito que deve ser corrigido.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - An apparent infinite recursive loop	Loop infinito que deve ser corrigido.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Apparent method/constructor confusion	Nomes de métodos que podem indicar erros na definição de construtores.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Array formatted in useless way using format string	Deve-se usar Arrays.asList(...) antes da formatação como Strings.	Ineficiência	Média	Corretude funcional	Adequação funcional
Correctness - Bad attempt to compute absolute value of signed 32-bit hashcode	Essa implementação retorna o valor negativo.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Bad attempt to compute absolute value of signed 32-bit random integer	Essa implementação retorna o valor negativo.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Bad comparison of nonnegative value with negative constant	Compara valor negativo com outro valor que já se sabe ser negativo.	Ineficiência	Alta	Utilização de recursos	Eficiência de performance
Correctness - Bad comparison of signed byte	Comparação de valores com limites diferentes.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Bad constant value for month	Passagem de parâmetro incorreto.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Bitwise add of signed byte value	Comparação de valores com limites diferentes.	Bug em potencial	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Bitwise OR of signed byte value	Indicativo de operações com valores incorretos.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Call to equals() comparing different interface types	Comparação de tipos diferentes.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Call to equals() comparing different types	Comparação de tipos diferentes.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Call to equals() comparing unrelated class and interface	Comparação de tipos diferentes.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Call to equals() with null argument	Comparação utilizando valores nulos.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Can't use reflection to check for presence of annotation without runtime retention	Boa prática no uso de reflexões.	Inconsistência de estilo	Média	Corretude funcional	Adequação funcional
Correctness - Check to see if ((...) & 0) == 0	Expressão que sempre retorna verdadeiro e deve ser revista.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Class defines field that masks a superclass field	Nomeação confusa de campos afeta a legibilidade do código.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Class overrides a method implemented in super class Adapter wrongly	Sobrecarga incorreta de método que pode levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - close() invoked on a value that is always null	Instrução incorreta que lança NullPointerException.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Dead store of class literal	Código inutilizado que deve ser excluído.	Ineficiência	Alta	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Deadly embrace of non-static inner class and thread local	Implementação incorreta que pode levar a criação de objetos que não serão utilizados e também não serão elegíveis para o garbage collector.	Bug em potencial	Ineficiência	Corretude funcional	Adequação funcional
Correctness - Don't use removeAll to clear a collection	Boa prática de programação que recomenda o uso de clear para limpar coleções.	Ineficiência	Alta	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Doomed attempt to append to an object output stream	Possível erro na utilização de arquivos.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Doomed test for equality to NaN	Checagem incorreta de valores aos quais são atribuídos o valor especial NaN.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Double assignment of field	Atribui o mesmo valor duas vezes. Pode indicar um erro de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Double.longBitsToDouble invoked on an int	Possível utilização incorreta do método long.BitsToDouble.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - equals method always returns false	Método que sempre retorna falso indica um possível erro de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - equals method always returns true	Método que sempre retorna verdadeiro indica um possível erro de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - equals method compares class names rather than class objects	Comparação incorreta de objetos de classes.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - equals method overrides equals in superclass and may not be symmetric	Possível erro na utilização da sobrecarga de métodos equals()	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - equals() method defined that doesn't override Object.equals(Object)	Quando o método equals() é utilizado ele deve sobrecarregar o método equals(Object)	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - equals() used to compare array and nonarray	Comparação de objetos com tipos diferentes.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - equals(...) used to compare incompatible arrays	Comparação de arrays incompatíveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Field only ever set to null	Campo que sempre tem o valor nulo. Deve-se verificar erros de programação.	Indicativo de erros de programação	Baixa	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - File.separator used for regular expression	Indicativo de má utilização de expressões regulares	Indicativo de erros de programação	Baixa	Corretude funcional	Adequação funcional
Correctness - Format string placeholder incompatible with passed argument	Erro na formatação dos resultados para exibição.	Bug	Média	Corretude funcional	Adequação funcional
Correctness - Format string references missing argument	Erro na formatação dos resultados para exibição.	Bug	Média	Corretude funcional	Adequação funcional
Correctness - hasNext method invokes next	Possível erro na utilização do método hasNext.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Illegal format string	Formato ilegal de String que leva à erro em tempo de execução.	Bug	Média	Corretude funcional	Adequação funcional
Correctness - Impossible cast	Cast impossível que pode levar a comportamentos inesperados.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Impossible downcast	Cast que sempre vai lançar a exceção ClassCastException.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Impossible downcast of toArray() result	Cast que sempre vai lançar a exceção ClassCastException.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Incompatible bit masks (BIT_AND)	Máscara de bits incompatível que pode levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Incompatible bit masks (BIT_IOR)	Máscara de bits incompatível que pode levar a comportamentos inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - instanceof will always return false	Deve-se investigar verificações instanceof() que vão sempre retornar falso.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - int value cast to double and then passed to Math.ceil	Possível passagem de parâmetros incorreta.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - int value cast to float and then passed to Math.round	Possível passagem de parâmetros incorreta.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Integer multiply of result of integer remainder	Possível confusão na precedência de operadores.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Integer remainder modulo 1	Possível confusão na utilização da operação módulo.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Integer shift by an amount not in the range 0..31	Operação bitwise que pode causar resultados indesejáveis.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Invalid syntax for regular expression	Deve-se verificar possíveis erros no uso de expressões regulares.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Invocation of equals() on an array, which is equivalent to ==	Compara a referência ao invés do valor.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Method assigns boolean literal in boolean expression	Indicativo de erro de programação que faz atribuição ao invés de comparação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Method attempts to access a prepared statement parameter with index 0	Passagem de parâmetro incorreta que pode levar a resultados indesejáveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Method attempts to access a result set field with index 0	Passagem de parâmetro incorreta que pode levar a resultados indesejáveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Method call passes null for nonnull parameter	Passagem de parâmetros nulos pode levar a resultados indesejáveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Method defines a variable that obscures a field	Variáveis com o mesmo nome de campos afeta a legibilidade do código.	Inconsistências de estilo	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Method does not check for null argument	Método não faz checagem de parâmetros nulos, o que pode levar a resultados indesejáveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Method doesn't override method in superclass due to wrong package for parameter	Possível erro na declaração dos tipos dos parâmetros	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Method ignores return value	Ignora o valor retornado pelo método, o que pode levar a resultados indesejáveis.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Method may return null, but is declared @NonNull	Métodos com tipos incompatíveis que podem levar a resultados indesejáveis.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Method must be private in order for serialization to work	Se o método não for privado ele será ignorado causando resultados indesejados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Method performs math using floating point precision	Deve-se considerar o uso de Double para não perder precisão.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - More arguments are passed that are actually used in the format string	Parâmetros desnecessários que devem ser excluídos.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - No previous argument for format string	Falta argumento para formatar a string, o que pode levar à MissingFormatArgumentExcepti on	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - No relationship between generic parameter and method argument	Parâmetros com tipos incompatíveis, o que pode levar a resultados indesejados.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Nonsensical self computation involving a field (e.g., x & x)	Operação desnecessária que pode indicar um erro de programação.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Non-virtual method call passes null for nonnull parameter	Passagem de parâmetro nulo para parâmetro anotado como não-nulo.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Null pointer dereference	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Null pointer dereference in method on exception path	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Null value is guaranteed to be dereferenced	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Nullcheck of value previously dereferenced	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Number of format-string arguments does not correspond to number of placeholders	Utilização incorreta de valores em Strings, pode levar a resultados inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Overwritten increment	Sobrescrita de valores que já foram atribuídos, pode indicar um possível erro de programação e levar a resultados inesperados.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Possible null pointer dereference	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Possible null pointer dereference in method on exception path	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Primitive array passed to function expecting a variable number of object arguments	Parâmetros com tipos incompatíveis, o que pode levar a resultados indesejados.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Primitive value is unboxed and coerced for ternary operator	Conversão de tipos incompatível que pode causar resultados indesejados	Bug em potencial	Média	Corretude funcional	Adequação funcional
Correctness - Random value from 0 to 1 is coerced to the integer 0	Geração incorreta de numerous aleatórios	Bug	Alta	Corretude funcional	Adequação funcional
Correctness - Read of unwritten field	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Repeated conditional tests	Testes condicionais repetidos que devem ser refatorados	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Self assignment of field	Atribuição de valores de um campo para ele mesmo, o que indica erros de programação.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Self comparison of field with itself	Comparação de um campo com ele mesmo, o que indica erros de programação.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Self comparison of value with itself	Comparação de um valor com ele mesmo, o que indica erros de programação.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Signature declares use of unhashable class in hashed construct	Assinatura de classe inconsistente, o que indica erro de programação e pode levar a comportamento inesperado.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - The readResolve method must not be declared as a static method.	Para que o método seja reconhecido, ele deve ser declarado como estático.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Uncallable method defined in anonymous class	Métodos que não são utilizados e devem ser excluídos	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Uninitialized read of field in constructor	Leitura de um campo que ainda não foi inicializado, o que pode levar a comportamento inesperado.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Uninitialized read of field method called from constructor of superclass	Leitura de um campo que ainda não foi inicializado, o que pode levar a comportamento inesperado.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Unnecessary type check done using instanceof operator	Checagem de tipo desnecessária que deve ser refatorada	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Unneeded use of currentThread() call, to call interrupted()	É recomendado utilizaro o método Thread.interrupted().	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Correctness - Unwritten field	Sempre retornará o valor padrão, o que indica erro de programação e pode levar a comportamento indesejado.	Indicativo de erros de programação	Alta	Corretude funcional	Adequação funcional
Correctness - Useless assignment in return statement	Valor que não é utilizado e deve ser removido.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Useless control flow to next line	Valor que não é utilizado e deve ser removido.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Correctness - Using pointer equality to compare different types	Comparação de tipos incompatíveis, o que pode levar a resultados indesejáveis.	Bug em potencial	Alta	Corretude funcional	Adequação funcional
Correctness - Value annotated as carrying a type qualifier used where a value that must not carry that qualifier is required	Utilização incorreta de modificadores.	Indicativo de erros de programação	Média	Corretude funcional	Adequação funcional
Correctness - Value is null and guaranteed to be dereferenced on exception path	Irá causa NullPointerException quando o código for executado.	Bug	Alta	Corretude funcional	Adequação funcional
Dataflow Anomaly Analysis	Verifica o uso anormal de variáveis que pode levar a erros	Indicativo de erros de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	Boa prática de programação que recomenda o uso de interfaces para declaração de coleções	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Design For Extension	Classes devem ser projetadas para a herança.	Inconsistência de estilo	Baixa	Modularidade	Manutenabilidade
Dont Use Float Type For Loop Indice	Float pode não ter a precisão desejada.	Futuro erro de programação	Média	Corretude funcional	Adequação funcional
Empty arrays and collections should be returned instead of null	Boa prática que força a checagem do retorno.	Inconsistência de estilo	Baixa	Utilização de recursos	Eficiência de performance

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Empty statements should be removed	Trechos vazios geralmente são introduzidos por engano e podem causar efeitos indesejáveis.	Indicativo de erros de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Exception handlers should provide some context and preserve the original exception	Deve-se preservar o contexto e a exceção para evitar possíveis confusões durante o debug.	Futuro erro de programação	Média	Analisabilidade e Testabilidade	Manutenabilidade
Exception types should not be tested using "instanceof" in catch blocks	Boa prática que recomenda o uso de vários blocos catch ao invés de capturar diversas exceções.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Exceptions should not be thrown in finally blocks	Pode mascara exceções anteriores e perder o contexto no qual as exceções foram lançadas.	Futuro erro de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Excessive Class Length	Classe muito grande que deve ser refatorada	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Excessive Public Count	Classe muito grande que deve ser refatorada	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Expressions should not be too complex		Ineficiência	Média	Analisabilidade	Manutenabilidade
Field names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade
Files should contain an empty new line at the end	Evita conflitos na utilização de softwares de controle de versão.	Inconsistência de estilo	Baixa	Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Files should not have too many lines	Arquivos com muitas linhas indicam classes ou métodos muito longas e com muitas responsabilidades que devem ser refatoradas.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
God Class	Classe muito grande e com muitas responsabilidades que deve ser refatorada.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
if/else/for/while/do statements should always use curly braces	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Immutable Field	Verifica a necessidade de se usar o modificador final em determinados campos	Inconsistência de estilo	Baixa	Modificabilidade	Manutenabilidade
Import Order	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Afeta a legibilidade e pode levar a comportamentos inesperados.	Futuro erro de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Inefficient String Buffering	Boa prática que recomenda que não se deve concatenar não-literais em StringBuffers	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Insufficient comment density	Deve-se determinar a quantidade aceitável de comentários que colaboram com o entendimento do código.	Inconsistência de estilo	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Insufficient String Buffer Declaration	Pode causar a mudança do tamanho da String várias vezes durante o tempo de execução, o que pode afetar a performance do programa.	Ineficiência	Média	Utilização de recursos	Eficiência de performance

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Integer Instantiation	Instanciação desnecessária	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Interface names should comply with a naming convention	Convenções de nomeação facilitam a legibilidade do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
IP addresses should not be hardcoded	O IP deve ser capturado dinamicamente para evitar a necessidade de recompilação e a necessidade de inserir o IP em ambiente de desenvolvimento.	Ineficiência	Média	Utilização de recursos	Eficiência de performance
Lambdas and anonymous classes should not have too many lines	Deve-se evitar classes anônimas e lambdas com muitas linhas de código, pois elas são usadas geralmente para inserir comportamento sem a necessidade de se criar uma classe específica.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Law Of Demeter	Boa prática de programação que evita o acoplamento entre classes ou objetos.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Left curly braces should be located at the beginning of lines of code	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Lines of code should not be too long	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Literal boolean values should not be used in condition expressions	Afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Local variable and method parameter names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Local variables should not shadow class fields	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Loggers should be "private static final" and should share a naming convention	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Long Variable	Nome muito longo que deve ser refatorado.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Loop counters should not be assigned to from within the loop body	Pode causar comportamento inesperado ao se modificar o contador dentro do loop.	Bug em potencial	Média	Corretude funcional	Adequação funcional
Loops should not contain more than a single "break" or "continue" statement	Boa prática de programação que evita loops muito complexos.	Ineficiência	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Magic Number	Convenção de codificação que afeta a legibilidade do código.	Inconsistência de estilo	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Malicious code vulnerability - Field is a mutable array	Array pode ter seu valor alterado por código malicioso.	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field is a mutable Hashtable	Hastable pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field isn't final and can't be protected from malicious code	Campo pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field isn't final but should be	Campo pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field should be both final and package protected	Campo pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field should be moved out of an interface and made package protected	Campo pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Field should be package protected	Campo pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Malicious code vulnerability - Finalizer should be protected, not public	Finalizador pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Malicious code vulnerability - Public static method may expose internal representation by returning array	Array retornado pode ter seu valor alterado por código malicioso	Bug	Muito Alta	Segurança	Integridade
Method names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Method parameters, caught exceptions and foreach variables should not be reassigned	Não se deve associar valor à parâmetros, pois esses valores serão perdidos, indica um erro de programação.	Indicativo de erros de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Methods should not be empty	Método vazio que deve ser removido.	Indicativo de erros de programação	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Methods should not be too complex	Métodos muito complexos devem ser refatorados.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Methods should not contain too many return statements	Métodos muito complexos devem ser refatorados.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Methods should not have too many lines	Métodos muito complexos e com muitas responsabilidades devem ser refatorados.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Modifiers should be declared in the correct order	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Naming - Avoid field name matching method name	Nome de variáveis iguais ao nome dos métodos devem ser removidos	Inconsistência de estilo	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Nested blocks of code should not be left empty	Blocos vazios que devem ser removidos.	Indicativo de erros de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
NPath complexity	Métodos muito complexos que afetam a manutenabilidade	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Null Assignment	Boa prática que recomenda que não se deve atribuir o valor null à variáveis	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Overriding methods should do more than simply call the same method in the super class	Consome recursos com uma operação ineficiente que deve ser evitada.	Ineficiência	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Package names should comply with a naming convention	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Performance - Could be refactored into a named static inner class	Definição desnecessária de classe interna que provoca a utilização desnecessária de recursos.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Could be refactored into a static inner class	Boa prática que recomenda a criação de classe interna com os modificadores static ineer	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Huge string constants is duplicated across multiple class files	Valores duplicados que provocam a utilização desnecessária de recursos.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Inefficient use of keySet iterator instead of entrySet iterator	Boa prática que recomenda o uso de entrySet para evitara utilização desnecessária de recursos.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Maps and sets of URLs can be performance hogs	Deve-se considerar o uso de java.net.URI para evitar problemas de perfomance	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Performance - Method allocates a boxed primitive just to call toString	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method allocates an object, only to get the class object	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method calls static Math class method on a constant value	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method concatenates strings using + in a loop	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes inefficient floating-point Number constructor; use static valueOf instead	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes inefficient new String() constructor	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes inefficient new String(String) constructor	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes inefficient Number constructor; use static valueOf instead	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method invokes toString() method on a String		Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Method uses toArray() with zero-length array argument	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Performance - Primitive value is boxed and then imMédiately unboxed	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Primitive value is boxed then unboxed to perform primitive coercion	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - The equals and hashCode methods of URL are blocking	Deve-se considerar o uso de java.net.URI para evitar problemas de perfomance	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Unread field	Campo que nunca é utilizado deve ser excluído.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Unread field: should this field be static?	Campo que nunca é utilizado deve ser excluído.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Unused field	Campo que nunca é utilizado deve ser excluído.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Performance - Use the nextInt method of Random rather than nextDouble to generate a random integer	Operação desnecessária que utiliza recursos que poderiam ser preservados.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
Public methods should throw at most one checked exception	Boa prática que recomenda que métodos públicos devem lançar exceções que foram checadas anteriormente.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Public types, methods and fields (API) should be documented with Javadoc	Boa prática que recomenda o uso do padrão Javadoc para documentação.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Redundant Field Initializer	Inicializador redundante que deve ser refatorado.	Ineficiência	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Redundant Modifier	Modificador redundante que deve ser refatorado.	Ineficiência	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Right curly brace and next "else", "catch" and "finally" keywords should be located on two different lines	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Security - A prepared statement is generated from a nonconstant String	Código SQL que se não for checado pode causar danos à aplicação.	Bug	Muito Alta	Integridade	Segurança
Security - Array is stored directly	Boa prática que previne a modificação indesejada de valores de arrays.	Bug	Muito Alta	Integridade	Segurança
Security - Empty database password	Indica que o banco de dados não está protegido.	Bug	Muito Alta	Integridade	Segurança
Security - Hardcoded constant database password	Senha do banco de dados não deve estar disponível no código fonte.	Bug	Muito Alta	Integridade	Segurança
Security - HTTP cookie formed from untrusted input	Formação de cookie que se não for checada pode causar danos à aplicação.	Bug	Muito Alta	Integridade	Segurança
Security - HTTP Response splitting vulnerability	Formação de resposta HTTP que se for exposta pode levar à exposição indesejada de informações.	Bug	Muito Alta	Integridade	Segurança
Security - JSP reflected cross site scripting vulnerability	Escreve parâmetros HTTP diretamente em uma saída JSP, o que pode levar à exposição indesejada de informações.	Bug	Muito Alta	Integridade	Segurança
Security - Method returns internal array	Permite que usuários modifiquem diretamente código que pode ser crítico	Bug	Alta	Integridade	Segurança
Security - Nonconstant string passed to execute method on an SQL statement	Pode executar código SQL malicioso.	Bug	Alta	Integridade	Segurança
Security - Servlet reflected cross site scripting vulnerability	Escreve parâmetros HTTP diretamente em uma página de erros, o que pode levar à	Bug	Alta	Integridade	Segurança

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
	exposição indesejada de informações.				
Short Class Name	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Short Variable	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Singular Field	Campo usado por um único método pode ser substituído por uma variável local	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Source code should be correctly indented	Convenções de codificação que facilitam o entendimento do código.	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Static Variable Name	Campo usado por um único método pode ser substituído por uma variável local	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
Strict Duplicate Code	Afeta a manutenção do código e indica alto acoplamento.	Indicativo de erros de programação	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
String literals should not be duplicated	DIficulta a refatoração, strings duplicadas podem ser substituídas por uma constante.	Ineficiência	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
String To String	Conversão desnecessária	Ineficiência	Baixa	Utilização de recursos	Eficiência de performance
String.valueOf() should not be appended to a String	Afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Strings literals should be placed on the left side when checking for equality	Previne que a exceção NullPointerException seja lançada.	Bug em potencial	Alta	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Switch cases should not have too many lines	Blocos switch muito complexos que devem ser refatorados.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Switch statements should end with a default case	Boa prática de programação defensive.	Inconsistência de estilo	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	Pode impactar negativamente na performance.	Ineficiência	Muito Alta	Utilização de recursos	Eficiência de performance
System.out and System.err should not be used as loggers	Deve-se usar loggers específicos.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
The default unnamed package should not be used	Convenções de nomeação facilitam o entendimento do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
The members of an interface declaration or class should appear in a pre-defined order	Convenções de codificação que facilitam o entendimento do código	Inconsistência de estilo	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Throwable.printStackTrace(...) should never be called	Deve-se usar o logger adequado.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Throws declarations should not be redundant	Declaração redundate que deve ser removida.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Too Many Fields	Clases com muitos campos que deve ser refatorada	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Too many methods	Classe muito grande que deve ser refatorada	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Try-catch blocks should not be nested	Afeta a legibilidade do código.	Inconsistência de estilo	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Unnecessary constructor	Construtor desnecessário que deve ser removido.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
Unnecessary Local Before Return	Return desnecessário que deve ser removido.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused formal parameter	Parâmetro desnecessário que deve ser removido.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused local variables should be removed	Variável desnecessária que deve ser removida.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused Modifier	Modificador desnecessário que deve ser removido	Ineficiência	Muito baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Unused Null Check In Equals	Checagem desnecessária que deve ser removida	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused private fields should be removed	Variável desnecessária que deve ser removida.	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused private method	Método desnecessário que deve ser removido	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Unused protected method	Método desnecessário que deve ser removido	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Use ConcurrentHashMap	Boa prática que recomenda o uso de uma interface específica para Map quando se está desenvolvendo aplicações concorrentes	Inconsistência de estilo	Baixa	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Use Locale With Case Conversions	Boa prática que recomenda o uso de Locale para conversões de caixa alta e caixa baixa	Bug em potencial	Alta	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade
Use String Buffer For String Appends	Verifica o uso de += para fazer a concatenação de strings	Ineficiência	Média	Modificabilidade e Analisabilidade	Manutenabilidade
Useless imports should be removed	Imports desnecessários que devem ser removidos	Ineficiência	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Useless parentheses around expressions should be removed to	Parenteses desnecessários que devem ser removidos	Ineficiência	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade

Regras	Impacto	Categoria	Severidade	Sub-característica associada	Característica associada
prevent any misunderstanding					
Utility classes should not have a public constructor	Utility classes não devem ser instanciadas, por isso não podem ter construtores públicos.	Indicativo de erros de programação	Baixa	Modificabilidade e Analisabilidade	Manutenabilidade
Variables should not be declared and then immediately returned or thrown	Variável desnecessária que deve ser removida.	Ineficiência	Média	Modificabilidade, Analisabilidade e Testabilidade	Manutenabilidade